GREGORY V. WILSON,
UNIVERSITY OF TORONTO

Extensible

Is an open, more flexible programming environment just around the corner?

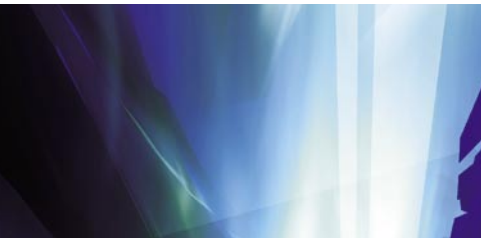
Programming for the 21st Century

In his keynote address at OOPSLA '98 (Object-Oriented Programming, Systems, Languages, and Applications), Sun Microsystems Fellow Guy L. Steele Jr. said, "From now on, a main goal in designing a language should be to plan for growth." Functions, user-defined types, operator overloading, and generics (such as C++ templates) are no longer enough: tomorrow's languages must allow programmers to add entirely new kinds of information to programs, and control how it is processed.

This article argues that next-generation programming systems can accomplish this by combining three specific technologies:

- Compilers, linkers, debuggers, and other tools that are frameworks for plug-ins, rather than monolithic applications.
- Programming languages that allow programmers to extend their syntax.
- Programs that are stored as XML documents, so programmers can represent and process data and meta-data uniformly.

These innovations will likely change programming as profoundly as structured languages did in the 1970s, objects in the 1980s, and components and reflection in the 1990s. To see why, we must first examine the shortcomings of the systems that programmers use today. Let's begin with two of the most popular: the Unix command line and Microsoft's COM (component object model).



Extensible Programming for the 21st Century

FRAMEWORKS

If success is measured by longevity, the Unix command line is the most successful programming system in history. Now more than 30 years old, it is still the favorite environment of many developers.

Conventional wisdom attributes its power to its "lots of little tools" philosophy: instead of writing everything from scratch, programmers can create complex data-processing pipelines with just a few keystrokes by combining wc, grep, and their kin. This is possible because these tools use a common data format and communication protocol. The format is a list of newline-terminated strings; the protocol is standard input, standard output, and exit(rc). Together, these conventions define a simple component model; any program that respects them can work with any other, no matter what language each is written in.

The Unix model works well for systems administration and simple text processing. It cannot, however, reliably handle data that isn't representable as a stream of records. In particular, programs, which are inherently tree-structured, can't accurately be parsed using regular expressions (the biggest guns in the command-line arsenal). Standard Unix tools therefore can't do things as simple as changing the names of variables.

The second weakness of command-line tools is that command-line flags are clumsy ways of specifying control flow. Everyone agrees that one-line programs are bad (well, everyone except die-hard Perl fans), but that is effectively what tools such as find require users to write. What is worse, every tool's command-line mini-language is different from every other's. Attempts to stick to simple on or off options lead to monsters like gcc, which now has so many flags that programmers are using genetic algorithms to explore them.[1]

Rather than abandon line-oriented tools, many programmers have turned their backs on problems they can't handle. Others have turned instead to component systems, such as Microsoft's COM. Rather than requiring programmers to represent everything as lists of strings, COM lets them pass data structures in memory. Instead of squeezing their intentions through the narrow filter of command-line mini-languages, programmers can then specify their desires using loops, conditionals, method calls, and all the other features of familiar languages. The result is that today's Windows developers can write programs in Visual Basic, C++, or Python that use Visual Studio to compile and run a program, Excel to analyze its performance, and Word to check the spelling of the final report.

Crucially, developers can also *extend* these products by writing plug-ins. For example, Visual Studio interacts with version control systems through a well-defined API. So long as someone cares enough to write the bridging code, any version control system that runs on Windows can be driven directly from Visual Studio's buttons and menus. A similar API allows the popular memory-checking tool Purify to be used in place of Visual Studio's own debugger, and so on.

Pluggability lies at the heart of most of today's sophisticated applications. Take Apache: everyone knows it's a Web server that sends HTML pages in response to HTTP requests. What is less widely known is that its plug-in system allows it to be a platform for other applications. By writing modules that inspect and modify requests as they pass through the server, developers can make Apache provide authentication services, version control, online games, and much more.

One of the great ironies of the early 21st century is that the programmers who build component systems for others are strangely reluctant to componentize their own tools. Compilers and linkers are still monolithic command-line applications: files go in, files come out, and the only way to control what happens in between is through command-line flags or embedded, vendor-specific directives. Programmers cannot invoke parsers, analyzers, or code generators selectively, or insert custom modules to change how programs are processed. (This is not an open source versus closed source issue: GCC, the GNU Compiler Collection, is no more a framework for plug-ins than commercial compilers.)

But why would anyone want to do these things? One

answer is given by SUIF (Stanford University Intermediate Format), a compiler that allows users to plug in their own optimization modules.[2] Developers can add new or improved optimizations to SUIF by writing a filter and adding it to the compiler's configuration. Doing so isn't simple, but as with Apache plug-ins, once one developer has done it, others can immediately share the benefits.

Now, consider your favorite debugger—or rather, compare it with your favorite editor. You can write macros for the latter; why not for the former? And why can't programmers include code in libraries to control how debuggers display the data structures those libraries create? Almost all debuggers display structures as a tree. This is adequate for shallow acyclic structures, but it's frustrating or misleading for large cyclic ones. If debuggers were programmable components, or if programmers could insert display callbacks in libraries, they could give users much more insight into what their programs were doing. For example, one of the reasons graphical debuggers such as DDD (Data Display Debugger, http://www.gnu.org/software/ddd/) have failed to catch on is that they display all data structures in terms of allocated blocks of memory. If graphical debuggers showed trees as trees, and queues as queues, programmers would be more likely to use them.

Making programming tools pluggable frameworks would be useful across the board, and I believe it is essential for supporting extensible languages. To see why, we must look at what extensibility means, and where it has been successful.

EXTENSIBLE SYNTAX

Programming languages often grow by formalizing and generalizing the best practices of their day. Well-nested `goto` statements become structured programming's conditionals and loops; records that are accessed only through companion functions become objects; functions that are identical except for data types become generics, and so on.

An extensible language is one that puts this power in everyone's hands, instead of reserving it for a standards committee. A *syntactically extensible language* allows programmers to define new forms by specifying what the new syntax looks like, and how it maps back to the language's primitives. A *semantically extensible language* allows programmers to define entirely new kinds of operations, or to change the behavior of built-in ones. C macros and C++ operator overloading are probably the most familiar examples of each kind of extensibility, although both are severely restricted.

Lisp and its child Scheme show how powerful wholehearted extensibility can be. Scheme programmers routinely use its *hygienic macros* to customize it for specific problem domains. For example, the macro definition:

```
(define-macro when
   (lambda (test . branch)
      `(if ,test
        (begin ,@branch))))
```

tells Scheme to translate:

```
(when (>= pressure limit)
   (open-valve 20)
   (close-valve))
```

into:

```
(if (>= pressure limit)
   (begin
      (open-valve 20)
      (close-valve)))
```

A less trivial example comes from the Java syntactic extender, a hygienic macro system for Java.[3] With it, programmers can define transformations to turn this:

```
check a.equals(b) throws NullPointerException;
```

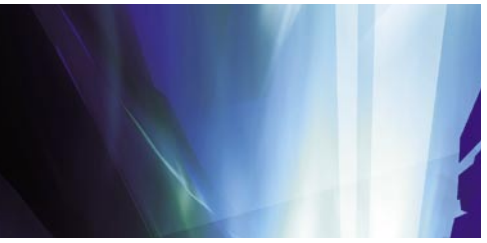
into this:

```
try {
   logCheck("a.equals(b) throws NullPointerException");
   a.equals(b);
   noThrowFailure();
} catch (NullPointerException e) {
   checkPassed();
} catch (Throwable t) {
   incorrectThrowFailure(t);
}
```

We assert without proof that programmers would create more unit tests if they could write them using the shorter, more readable, form.

Language extensibility has been around for years, but is still an academic curiosity. Three problems stand in the way of its general adoption: its unfamiliarity, the absence of support for it in mainstream languages, and the cognitive gap between what programmers write and what they have to debug. The first is slowly being eroded by code generators such as XDoclet,[4] and by the wizards used to generate manifests and other boilerplate in Enterprise JavaBeans[5] and .NET.[6] The second problem is a result of the first: as programmers become more comfortable with program transformation tools, language support should inevitably follow. Java 1.5, for example, allows programmers to annotate their programs, so that:

```
public class CoffeeOrder {
   @Remote public Coffee [] getPriceList() {
      ...
   }
```



Extensible Programming for the 21st Century

```
   @Remote public String orderCoffee(String name,
   int quantity) {
      ...
   }
}
```

can be transformed into:

```
public interface CoffeeOrderIF extends java.rmi.Remote {
   public Coffee [] getPriceList()
      throws java.rmi.RemoteException;
   public String orderCoffee(String name, int quantity)
      throws java.rmi.RemoteException;
}
public class CoffeeOrderImpl implements CoffeeOrderIF {
   public Coffee [] getPriceList() {
      ...
   }
   public String orderCoffee(String name, int quantity) {
      ...
  }
}
```

This leaves the third, and thorniest, problem: the cognitive gap between what programmers write and what they have to debug. This gap is why so many programmers dislike wizards and other code generators. If the generated code misbehaves, the programmer must:

- Abandon it and write the code by hand (just as if the high-level generator didn't exist).
- Edit and debug the code generated by the wizard (which is invariably more convoluted than what the programmer would have written).
- Tweak the source code blindly in the hope of producing the output that does what the programmer originally wanted.

The first option leads many programmers to believe that code generators are a waste of time; the second, to unmaintainable spaghetti; and the third, to despair.

The only scalable way to bridge this gap is to let programmers control how linkers, debuggers, and other tools handle generated code. When programmers add something to a language, they should be able to plug a module into the compiler to tell it how to generate code for the new feature, and another module into the debugger to tell it how to display uses of that feature.

Rather than today's "passive" libraries, containing only code and data for the final program, tomorrow's programmers will work with *active libraries*.[7] These will contain not only content to be included in the final application, but also instructions telling processing tools how to analyze, optimize, and debug that content. As already mentioned, only a small minority of programmers will need to create such libraries to make everyone else more productive, just as only a few need to create plug-ins for Apache and Visual Studio today.

Pluggable frameworks and active libraries will help make extensibility accessible, but they will not be enough on their own. To see why, we must take a closer look at why extensibility has been so successful in Lisp and its offspring, but have failed to catch on elsewhere.

MODELS AND VIEWS

Programmers have been joking for decades that Lisp stands for "lots of irritating single parentheses." Behind those jokes lies a profound idea: in Lisp, programs and data are both represented as nested s-expressions. This encourages Lisp programmers to think of programs *as* data and to manipulate them the same way they manipulate everything else.

Most programmers turned up their noses at Lisp's prefix notation and parentheses. Those same programmers, however, have raced to adopt XML. Originally intended for representing data, XML has been pressed into service as a medium for programs as well. The best-known example of this is probably JSPs (Java server pages), which allow programmers to embed fragments of Java programs in HTML. When a user requests a JSP from a Web server, the server compiles the JSP into a pure Java servlet, then compiles and runs the servlet and sends its output to the user. For example:

```
<HTML>
<BODY>
<TABLE BORDER=2>
<%
```

```
 for (int i = 1; i<=10; i++) {
%>
   <TR>
   <TD><%= i %></TD>
   <TD><%= i*i %></TD>
   </TR>
<%
 }
%>
</TABLE>
</BODY>
</HTML>
```

becomes:

```
class MyPage extends Servlet {
 public String handleRequest(
   ServletContainer sc,
   Request req,
   Response res
 ) {
   PrintWriter out = res.getWriter();
   out.println("<HTML>");
   out.println("<BODY>");
   out.println("<TABLE BORDER=2>");
   for (int i=1; i<=10; i++) {
     out.println("<TR>");
     out.println("<TD>" + (i) + "</TD>");
     out.println("<TD>" + (i*i) + "</TD>");
     out.println("</TR>");
   }
   out.println("</TABLE>");
   out.println("</BODY>");
   out.println("</HTML>");
 }
}
```

This approach has much to recommend it, but it also has some serious weaknesses. On the positive side, JSPs allow programmers and graphic designers to see their code *in situ*. JSPs also support extensibility: JSP libraries predefine many tags for common actions, but users can extend these libraries by defining custom tags, and telling the JSP system which methods of which classes to invoke when those tags are used.[8]

On the negative side, JSPs are a prime example of the cognitive gap discussed earlier. If something goes wrong in a complex JSP, its author must wade through pages of machine-generated Java, or reverse engineer the translation process to come up with a fix.

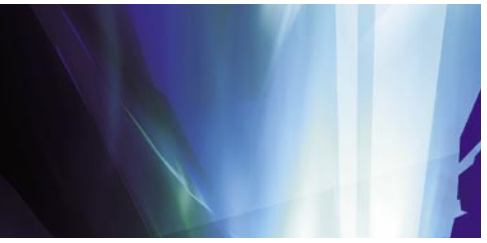
Ant, from the Apache Software Foundation, is another hybrid system that shares these strengths and weaknesses. Developed as a platform-independent replacement for Make, Ant has become the *de facto* standard build tool for Java. Ant uses XML, rather than a custom syntax, so off-the-shelf XML tools can be used to create, inspect, and modify Ant build files (figure 1). However, Ant shares a weakness with many hybrid systems: much of what is important in an Ant file isn't visible at the XML level. Variable references such as ${src} and ${dist}/lib/MyProject-${DSTAMP}.jar are much easier for human beings to read and write than their deeply nested XML equivalents would be, but are invisible to XML processing tools. To analyze and manipulate Ant files, programs have to perform a second, nonstandard round of parsing.

XSL, which is used to translate XML into HTML and other text formats, shares this flaw. XSL is a more-or-less declarative language, based on a match/replace execution model with forall and conditional constructs. It is cleaner

Example Ant File

```
<project name="MyProject" default="dist" basedir=".">

  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init" description="setup">
    <tstamp/>
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile">
    <mkdir dir="${dist}/lib"/>
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
basedir="${build}"/>
  </target>

  <target name="clean">
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

FIG 1



Extensible Programming for the 21st Century

than alternatives (such as transforming XML in Java or Perl), and there are even source-level debuggers, but once again, much of what is important in an XSL program is invisible to an XML parser. For example, consider this simple XSL program:

```
<BODY bgcolor="{/Member/FavoriteColor}">
  Welcome <xsl:value-of select="/Member/Name"/>!
  <xsl:if test="/Member/@level='gold'">
    Our special offer to gold members today is now open.
  </xsl:if>
  Your phone numbers are:
  <TABLE border="1" width="25%">
    <TR><TH>Type</TH><TH>Number</TH></TR>
    <xsl:for-each select="/Member/Phone">
      <TR>
        <TD><xsl:value-of select="@type"/></TD>
        <TD><xsl:value-of select="."/></TD>
      </TR>
    </xsl:for-each>
  </TABLE>
</BODY>
```

The first if test checks an individual's membership level. The condition in the test is invisible to XML parsers: all they see is a string, which has to be handed to a special-purpose tool for analysis.

If mixed representations are so clumsy, why do language designers perpetrate them? The answer is simple: "pure" XML is unpleasant to read and write. The more explicit the nesting becomes in XML, the harder it is for people to make sense of it.

But why should they have to? Why, in the early 21st century, do programmers still insist that their tools have to draw exactly one glyph on the screen for each byte in their source files? No one expects AutoCAD or Microsoft Word to do this; even grizzled old Unix fanatics don't expect to be able to open a relational database with Vi or Emacs. One of the great ironies of the early 21st century is that secretaries can easily put organizational charts or cubicle floor plans in e-mail messages, but the programmers who made that possible can't put class diagrams in their code.

We believe that next-generation programming systems will most likely store source code as XML, rather than as flat text. Programmers will *not* see or edit XML tags; instead, their editors will render these models to create human-friendly views, just like Web browsers and other WYSIWYG editors. For example, a program stored on disk like this:

```
<doc>Only replace below threshold</doc>
<cond>
 <test>
  <compare-expr operator="less">
   <field-expr field="age">
    <evaluate>record</evaluate>
   </field-expr>
   <evaluate>threshold</evaluate>
  </compare-expr>
 </test>
 <body>
  <invoke-expr method="release">
   <evaluate>record</evaluate>
  </invoke-expr>
 </body>
</cond>
```

would be viewed *and edited* like this:

```
// Only replace below threshold
if (record.age < threshold) {
  record.release();
}
```

Crucially, code will *not* be stored as uninterpreted CDATA within XML documents and programmers will not see (much less type in) XML tags. Such a representation would have all the disadvantages of JSPs, Ant, and other hybrid systems, without bringing any tangible benefits. Instead, XML will represent the program's deep structure. Only time and experimentation will tell whether this turns out to be something like an annotated syntax tree or something more abstract.

Using XML to separate software models from software views would bring several benefits. First, it would make languages more extensible. Programs stored as XML would be easier to process than ones stored as collections

of arbitrary ASCII tokens. In particular, programmers would be able to apply XSL and other tools to them—tools that will be as familiar to tomorrow's programmers as regular expressions are to today's.[9]

Second, it would simplify the construction of active libraries by letting programmers mark up their code to indicate which sections are intended for which tools, all in one file and all using one notation. Programmers would also be able to embed arbitrary content in their code, including mathematics (using MathML), class diagrams (using scalable vector graphics, or SVG), and all the meta-data that CASE (computer-aided software engineering) tools require. Donald Knuth's dream of "literate programming" could therefore be realized.[10]

Program with One Layer of Interpretation Removed

```
/  *  *  \r \n     *     T  h  i  s     c  l  a
s  s     p  r  i  n  t  s     <  e  m  >  o  d
d     n  u  m  b  e  r  s  <  /  e  m  >  .  \r
n     *     S  e  e     t  h  e     <  a     h
r  e  f  =  "  {  @  d  o  c  R  o  o  t  }  /
c  o  p  y  r  i  g  h  t  .  h  t  m  l  "  >
C  o  p  y  r  i  g  h  t  <  /  a  >  .  \r \n
   *     @  a  u  t  h  o  r     G  r  e  g
W  i  l  s  o  n  \r \n     *     @  v  e  r  s
i  o  n     1  .  2  \r \n     *  /  \r \n \r \n
p  u  b  l  i  c     c  l  a  s  s     O  d  d
s     {  \r \n        p  u  b  l  i  c     s  t
a  t  i  c     v  o  i  d     m  a  i  n  (  S
t  r  i  n  g  [  ]     a  r  g  s  )     {
   \r \n                 f  o  r     (  i  n  t
i  =  0  ;     i  <  1  0  ;     +  +  i  )
{        \r \n                       i  f     (  i
   %     2     =  =     0  )     {  \r \n
                    S  y  s  t  e  m  .  o  u  t
.  p  r  i  n  t  l  n  (  i  )  ;  \r \n
               }  \r \n                 }  \r \n
}  \r \n  }  \r \n
```

FIG 2

Finally, programmers could stop arguing about where curly braces should go, since they would be able to customize their views of software without modifying the underlying model. For example, a programmer could easily choose to view the previous code fragment as this:

```
28. if (record.age < threshold) /* Replace below threshold */
29. {
30.     record.release();
31. }
```

or even this:

```
;;; Replace below threshold
(if (< (record 'age) threshold)
   (record 'release))
```

without altering the underlying representation. (This is almost possible today with Microsoft .NET. As many have observed, it is the world's first "skinnable" programming system: C#, VB.NET, and other languages have the same semantics, built-in types, and libraries, and differ only in "superficial" details of syntax.)

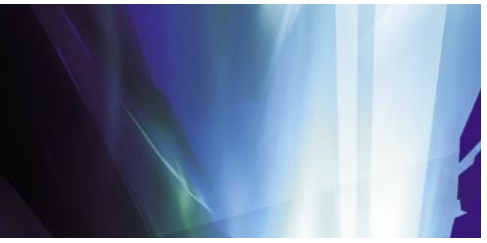
Yes, this could all have been done 20 years ago using s-expressions. As attractive as parenthesized lists are, however, they failed to win programmers' hearts and minds. In contrast, it has taken XML less than a decade to become the most popular data format in history. Every large application today can handle it; every programming language contains libraries for manipulating it; and every young programmer is as familiar with it as the previous generation was with streams of strings. S-expressions might have deserved to win, but XML has.

GETTING THERE

There are dozens of technical challenges to solve over the next decade to make extensible programming systems a success. The biggest challenge, though, will be social. Tell programmers that you're going to completely re-architect their tools, and they nod their heads. Tell them that you're going to store programs in something other than flat ASCII, and they start to squawk.

Many swear they will never use a system that doesn't store their programs "as they really are," conveniently ignoring the fact that it takes several hundred thousand lines of device drivers, operating system, and graphical interface to turn magnetic spins on a disk into characters on a screen. Remove just one layer of interpretation, and programs look like that shown in figure 2. Remove another, and we would have a table of numeric character codes; another, and we would have a stream of bits. It is therefore hypocritical to object to *adding* another layer (unless the person raising the objection still uses a text-only Web browser such as Lynx).

It also ignores the fact that fewer and fewer documents are stored as byte-per-character ASCII; increasingly, documents use some character encoding scheme to represent



Extensible Programming for the 21st Century

Unicode, which editors then interpret on the fly. As Unicode and XML documents become ubiquitous, so too will editors that apply style sheets and other transformations dynamically to make heterogeneous content comprehensible. Eventually, these will most likely merge with program editors, which cross-reference method calls and critique coding style in realtime.

But if we're going to leave flat ASCII behind, why replace it with XML? The IL (intermediate language) used in Microsoft .NET, Eclipse's AST, or serialization of the compiler's internal data structures all have much to recommend them. Programmers are already used to working with XML, however, and have powerful off-the-shelf tools to manipulate it. More important, most of the other information they would want to put into programs is available as XML. Like Inglish speling, XML is here to stay.

What about security? If anyone and everyone can inject new code into the compile/link/debug tool chain, what's to stop a malicious (or unlucky) developer from creating something that generates insecure or damaging code? The answer is, "Nothing." Current research on proving compiler optimizations or source refactorings sound may eventually provide the kind of security that bytecode verifiers bring to Java at runtime. Until then, users will have to be careful about what they download and use—just as they are with Web server extensions, Web browser plug-ins, and expansion packs for games.

Finally, language features interact in subtle ways—if every high school student with a bright idea is allowed to add a pet feature to the language, won't the result be incomprehensible gibberish?

"Compared with what," you ask? Right now, programmers must mentally parse several function calls to understand that a piece of Java is trying to match a string with a regular expression, or adding corresponding elements of two lists. Operator overloading and other kinds of extensibility can make these things much easier to understand (though their abuse by some C++ programmers is proof that the opposite is also true); extending extensibility's reach might finally give us readable notations for concurrency, database access, and so on.

Most programmers won't build extensions; they will pick and choose among the extensions that others have built. The resulting "free market" will give extensions that are readable a chance to beat out those that are not. There is no guarantee that the best will win (mostly because of the difficulty of getting three programmers to agree on what is best in any situation), but this approach should prove superior to waiting for standards committees to reach consensus, only to find that they have produced yet another camel.

IN DEFENSE OF MONOPOLIES

The changes proposed here could happen incrementally: editors and compiler front ends could appear first, leaving downstream tools to catch up later. None of these advances, however, will have full impact until all of them are in place. Given the number of players involved, it seems impossible for such a "big bang" to happen…

…unless, of course, everything is owned by a single vendor. Microsoft could easily have decreed that VB.NET source files would be XML documents; Wolfram Research or MathWorks could do the same in the next releases of Mathematica or MATLAB, respectively, and so on.

In the case of numerical languages such as Mathematica and MATLAB, using XML for storage would allow programmers to put real mathematical notation directly into their source files. In the case of .NET, XML storage would allow tools to embed meta-data directly in code. Right now, half of the average .NET program is deployment descriptors, property files, and so on. Programmers need conditionals, for-each iteration, and reuse in those descriptors, so their notations are slowly growing to include everything that a real programming language does. Sooner or later, vendors will stop trying to solve problems with "little languages," and start using full-strength solutions everywhere.[11]

CONCLUSION

New XML-based languages are already appearing (e.g.,

Mozart, SuperX++, o:XML),[12] but nothing is preventing development of extensible, XML-based versions of Java, Python, and C#. Designers are already looking at adding XML to these languages as a primitive data type.[13,14] It would be a natural next step to allow programmers to embed XML documentation (instead of using pseudo-HTML hacks such as JavaDoc), then use it for meta-data (such as compiler directives) and so on.

These are useful innovations, but they do not necessarily give programmers new ways to innovate themselves. The changes described in this article would. Instead of treating each new idea as a special case, they would allow programmers to say what they want to, when they want to, as they want to. The result would be systems that are simultaneously more sophisticated and easier to understand.

Of course, none of this is inevitable. We could still be writing and viewing programs a byte at a time in 2010, streaming them through command-line compilers, cursing our debuggers for their recalcitrance, and scratching our heads as we try to translate the obvious meaning of what's scrawled on the whiteboard into nested method calls. But do you really believe that will happen? Do you really believe that ours will be the only documents that aren't marked up, that can't contain heterogeneous content, that aren't processed by extensible frameworks? Alvin Toffler once said that the future always arrives too soon, and in the wrong order. Speaking as someone who has typed in a lot of bytes in the past 20 years, and cursed a lot of debuggers, extensible programming systems are a future that can't possibly arrive too soon. Q

REFERENCES

1. Ladd, S. R. 2003. *An evolutionary analysis of GNU C optimizations*; see http://www.coyotegulch.com/.
2. SUIF: see http://suif.stanford.edu/.
3. Bachrach, J., and K. Playford. 2001. *The Java Syntactic Extender;* see http://www.ai.mit.edu/~jrb/jse/jse.pdf.
4. XDoclet: see http://xdoclet.sourceforge.net/.
5. Herrington, J. 2003. *Code Generation in Action*. Greenwich, CT: Manning.
6. Dollard, K. 2004. *Code Generation in Microsoft .NET*. Berkeley, CA: Apress.
7. Czarnecki, K., Eisenecker, U., Gluck, R. Vandevoorde, D., and Veldhuizen, T. L. 1998. Generative programming and active libraries. In *Proceedings of Generic Programming '98*, Lecture Notes in Computer Science 1766, Springer-Verlag Telos.
8. Patzer, A. 2002. *JSP Examples and Best Practices*. Berkeley, CA: Apress.
9. See reference 6 (Dollard) for a discussion of the pros and cons of using Microsoft .NET's CodeDom for code generation.
10. Knuth, D. E. 1992. *Literate Programming*. Cambridge University Press.
11. See CONS (http://www.dsmit.com/cons/) and SCons (http://www.scons.org/) for examples of code-oriented build tools. Interestingly, the creator of Ant, James Duncan Davidson, has written: "*If I knew then what I know now, I would have tried using a real scripting language, such as JavaScript via the Rhino component or Python via JPython, with bindings to Java objects which implemented the functionality expressed in today's tasks. Then, there would be a first-class way to express logic and we wouldn't be stuck with XML as a format that is too bulky for the way that people really want to use the tool.*" http://x180.net/Articles/Java/AntAndXML.html.
12. Mozart: see http://mozart-dev.sourceforge.net/; SuperX++: see http://xplusplus.sourceforge.net/; o:XML: see http://www.o-xml.org.
13. ECMA-357: ECMAScript for XML: see http://www.ecma-international.org/publications/standards/Ecma-357.htm.
14. Meijer, E. Schulte, W., and G. Bierman. Programming with circles, triangles, and rectangles; see http://www.research.microsoft.com/~emeijer/Papers/XML2003/xml2003.html.

ACKNOWLEDGMENTS

This article grew out of an earlier one I wrote for *Doctor Dobb's Journal* ("XML-based programming systems," March 2003). I would like to thank Kimanzi Mati, Simon Peyton-Jones, Jim Larus, Mike Donat, Paul Prescod, Todd Veldhuizen, Mathew Zaleski, and Mark Mitchell for comments on it, and Ted Leung, Irving Reid, Charlie Reis, Dave Thomas, Matt Warren, and Jim Maurer for comments on the revisions.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

GREG WILSON holds a Ph.D. in computer science from the University of Edinburgh and has worked in high-performance scientific computing, data visualization, computer security, and software engineering. He is the author of *Practical Parallel Programming* (MIT Press, 1995), a contributing editor with *Doctor Dobb's Journal*, and an adjunct professor in computer science at the University of Toronto.