Tag Libraries as Fifth Generation Languages

Mark Cyzyk*

Scholarly Communication Architect, Library Digital Programs Group, The Sheridan Libraries, Milton S. Eisenhower Library, Johns Hopkins University, 3400 North Charles Street, Baltimore, MD 21218, USA

Abstract: This essay weighs the notion that Web-based tag libraries represent the fifth generation of computer languages. A brief survey of the history of computer languages is provided as is an indication of why it is useful to conventionally categorize computer languages into "generations". The essay argues that the current conventional ascription of generations is incorrect and that it should be replaced with one based on what the author terms "the principle of abstraction".

Keywords: Web-based tag libraries, generations of computer languages, scripting languages, history of computer languages, fifth generation languages.

TAG LIBRARIES AS FIFTH GENERATION LANGUAGES

Web-based tag libraries represent the fifth generation of computer languages.

In this essay I will explain what tag libraries are, what the generations of computer languages are and why they are important, how our conventional list of generations of computer languages has gotten off-track and is incorrect, and why tag libraries should properly be construed as falling into a fifth generation of computer programming languages.

What Tag Libraries Are

Tag libraries are a form of Web-based programming language, similar to the tag-based HTML, yet much more powerful. Unlike HTML, which is merely a markup language used to describe to a Web browser how content should be rendered, tag libraries provide powerful programming constructs similar to many other programming languages and thus can be used to create full-fledged Web applications with complex business logic.

For example, consider the following snippet of HTML:

```
<h3>3/1/2008</h3>
```

This simply outputs a string to the browser window, and that string is formatted as an HTML H3 header. This output is hard-coded, static.

Now consider the following snippet of ColdFusion Markup Language (CFML), a Web-based tag library:

```
<cfset thisDate = dateFormat(Now(),
'mm/dd/yyyy')>
<cfoutput>
  <cfif dateCompare(thisDate,
'4/15/2008') IS 0>
      <h2>#thisDate#</h2>
  <cfelse>
      <h3>#thisDate#</h3>
  </cfif>
</cfoutput>
```

This code supplements the HTML and adds intelligence to it while retaining the simplicity and elegance of a tag-based language. The code first sets a variable, "thisDate", equal to the mm/dd/yyyy-formatted value of the current system date. It then compares that date with April 15, 2008. If the two are equal, or if thisDate is greater than April 15, 2008, the value of the thisDate variable is output in an HTML H2 header. If not, then the value of the thisDate variable is printed in an HTML H3 header.

The output is not hard-coded; it is dynamically generated based on program logic.

All tag libraries provide the facility to set and read variables, provide a full set of conditional logic constructs, provide string and number formatting functions, and provide the facility to read and write to external databases. In short, they provide the basic functionalities of third generation languages and scripting languages, but often without the complexity associated with such languages. Thus they are often considered to exist at a higher level of abstraction than, say, 3GL or scripting languages.

Examples of this type of programming language include the venerable ColdFusion Markup Language (CFML) used in the illustration above and the relatively recent JavaServer Pages Standard Tag Library (JSTL).

The Traditional "Generations" of Languages

The history of programming languages is long, interesting, and instructive not only in the sense that by having a solid understanding of the evolution of any human artifact and practice leads to enlightenment about who we are and how we got here, but also in the sense that by looking at how changes were brought about in the programming world we gain an understanding of where programming languages might lead in their future evolution.

The first generation language consisted of machine code. Programming during this generation consisted of issuing explicit instructions to a particular processor that resulted in

*Address correspondence to this author at the Scholarly Communication Architect, Library Digital Programs Group, The Sheridan Libraries, Milton S. Eisenhower Library, Johns Hopkins University, 3400 North Charles Street, Baltimore, MD 21218, USA; Tel: (410) 516-0819; Fax: (410) 516-6229; E-mail: mcyzyk@jhu.edu

1874-1339/08

swapping values among memory locations. This process was not only relative to the processor for which one was programming, but it was also extremely error prone due to the great distance between machine code and human readable natural language. Clearly, a less tedious method of programming was called for.

The second generation of language was assembler. Assembler, like machine code, was written relative to a particular processor. However, assembler represented the first abstraction away from the specifics of the hardware in that it provided macros and mnemonic devices that could be used as shortcuts for machine code. Even so, assembler remained obscure and convoluted, and programming with it was only slightly less tedious than programming in machine code.

The third generation of languages (3GL) was when true competition and design of computer languages began; it was during this stage in the evolution of programming languages that design and diversity flourished. Examples of 3GLs include: C; C++; Pascal; Smalltalk; Lisp; Ada; Fortran; Basic; and Cobol. Oftentimes, a 3GL was designed to meet the programming needs of those working on a specific problem or within a specific industry. For example, Cobol was largely used in the business world, Fortran was (and still is) used primarily for scientific programming; Ada is used for military applications; and Lisp was (and remains) a good choice for artificial intelligence programming.

The fourth generation of languages includes the Structured Query Language (SQL), macro languages, and other specialized, languages. The fourth generation has always been a sort of catch-all for any higher-level language that could not easily be classified as a true 3GL.

Why Talking of Generations of Languages is Important

Categorizing computer languages by "generation" is important because it allows us to not only trace the evolution of these unique human artifacts, but to clarify the conceptual specificity of a language and the distance from natural language to which it stands. Categorizing computer languages by generations allows one to describe, quickly and succinctly, just how difficult a programming task will typically be to achieve in a particular set of languages. And this, of course, has ramifications for language choice when working in a particular domain of activity. For example, if one is writing device drivers one will need to choose a language designed for low-level interaction with hardware. However, if one does not need to implement such interaction, a low-level language with all its complexities is not only unneeded, but would be an impediment. Talking of generations of languages is important because, in a very practical sense, having languages categorized by complexity and having them properly situated in groups relative to their distances from the machine facilitates language choice and aligns those choices with the practical tasks required by any given programming project.

Generalizing on the history of programming languages, we arrive at the following principle: A new generation of computer language represents an abstraction, away from the complexity and specificity of machine code and toward more understandable and powerful natural language constructs.

Why Traditional Demarcations of Generations are Wrong

It can be seen, when looking at the traditional demarcations of computer languages, that they illustrate this principle for the most part. Certainly the first three generations clearly illustrate a move away from early, primitive machine code to more natural language constructs. However, insofar as the fourth generation of languages is comprised of a hodgepodge of otherwise unclassifiable languages, this breaks with the principle of abstraction. In fact, there is a type of language that deserves its own generational classification at the 4GL, but would typically be included as a 3GL. This type of language includes the so-called scripting languages. Scripting languages are interpreted languages, typically used as "glue" languages to tie together disparate modules written in other lower-level languages [1]. They are, though, also used to create full-fledged applications. Such languages as Python, Perl, PHP, ASP, TCL, and Ruby fall into this category. And in addition to being interpreted, their syntax is oftentimes much more like natural language than 3GLs. So scripting languages are different enough from 3GLs that they deserve their own generational demarcation, and so should be placed directly above the 3GLs, at the fourth generation.

Tag Libraries as The Fifth Generation

Being true to the principle of abstraction, we can and should then place tag libraries at the fifth generation. Insofar as they abstract away the complexities of even the relatively simple scripting languages at the fourth generation, they deserve their own generational demarcation at the fifth generation.

Comparing tag libraries with scripting languages, their nearest neighbor in the evolutionary chain, is particularly instructive. For example, compare what's involved in performing a simple database query and displaying the result in PHP, a scripting language, and JSTL and CFML, two tag libraries. First, PHP:

```
<?php
$connection = mysql_connect( "local-
host", "someuser", "somepassword" );
if ( ! $connection )
  die( "Couldn't connect!" );
mysql_select_db( $db, $connection );
  or die( "Couldn't open DB!" );
$result = mysql_query( "SELECT * FROM
TBLMAIN" );
while ( $row = mysql_fetch_row( $result
) )
  {
  foreach ( $row as $field )
      print "$field<br>\n";
  print"<p>\n";
  }
mysql_close( $connection );
?>
```

```
Then JSTL:

<sql:setDatasource
driver="com.mysql.jdbc.Driver"
url="jdbc:somedatabase:." user="sa"
password="somepassword"
var="mydatasource">

<sql:query var="result" sql="SELECT *
FROM TBLMAIN" />

<c:forEach items="$result.rows"
var="row">

   <c:out value="${row.firstname}"/><br>

   <c:out value="${row.lastname}"/>

   <p>

</c:forEach>

And in CFML this becomes even simpler:

<cfquery datasource="someDSN" pass-
word="somepassword" name="results">

SELECT * FROM TBLMAIN

</cfquery>

<cfoutput query="results">

$firstname<br>

$lastname

<p>

</cfoutput>
```

The key here is to note that the PHP example performs several relatively low-level functions to gain a useable connection to the database and to issue a query. It first uses the "mysql_connect" function to create a connection. It then uses the "mysql_select_db" function to select the correct database. The "mysql_query" function is used to actually pass an SQL query through to the database. And finally, the "mysql_close" function is used to close the previously-opened connection to the database.

Both JSTL and CFML largely hide this complexity from the programmer. JSTL encapsulates database connectivity functions in a single tag, the sql:setDataSource tag. ColdFusion assumes that a datasource name (DSN) has already been set up *via* the ColdFusion administrative utility. In both cases, the syntax is much easier to read than that of the more verbose PHP. The tag libraries hold to the principle of abstraction: The programmer does not need to know about how database connections are made, only that one is available for him to pass his SQL query into [2].

It is also interesting to note how browser output is handled. In PHP an explicit "print" statement must be issued in order to output any HTML to the screen. The HTML code must therefore be properly wrapped in quotation marks as required by the print statement, which itself must be terminated with a semi-colon. In JSTL and CFML, however, the situation is just the opposite. Here the assumption is that all output will occur on an HTML page, so HTML output does not need to be wrapped in any special syntax – rather, it is the existence and display of variable values that must be properly specified. JSTL does this *via* the <c:out /> tag; CFML wraps any code outputting the contents of variables in <cfoutput> tags. Such output happens, therefore, inline with the rest of a normal HTML page. Using these tag libraries, it's as if they were merely extensions, albeit very powerful extensions, to HTML.

In both cases, JSTL and CFML, the total code required to complete the task is much shorter than that required by PHP. This has implications with respect to developer productivity [3], especially when consideration is taken of the hundreds of database queries that may be required by the average application.

Consider another example, that of programmatically retrieving a page from the Web and displaying it to the client browser:

```
In ASP.NET [4]:

<%

Dim objWinHttp

Dim strHTML

Set objWinHttp =
Server.CreateObject("WinHttp.WinHttpRequ
est.5")

objWinHttp.Open "GET",
"http://someserver.com/somedirectory/som
efile.html"

objWinHttp.Send

strHTML = objWinHttp.ResponseText

Set objWinHttp = Nothing

%>

<%= strHTML %>

In JSTL this same functionality is im-
plemented by the following:

<c: import
url="http://someserver.com/somedirectory
/somefile.html" />
```

This single line of code not only retrieves the contents of the page specified by the URL, it also automatically outputs it to the client browser without requiring any other explicit instruction to do so.

Historical Notes

Princeton historian Michael Mahoney captures the moment when the movement between generations of computer languages began. In contrast to the previous work of engineers in creating more and more function libraries for assemblers, he notes: "The first high-level programming languages, perhaps most famously FORTRAN in 1957, followed over the next three years by LISP, COBOL, and ALGOL, took a quite different approach to programming by differentiating between the language in which humans think about problems and the language by which the machine is addressed." [5] It is this distinction that serves as the impetus for the creation of generations of computer languages. How can we solve problems and get work done without having to directly address the machine in its native language? Thus begins the history of computer languages.

This history is detailed in Jean Sammet's monumental 1969 study, *Programming languages: History and fundamentals* [6] as well as in the proceedings of both the first ACM SIGPLAN History of Programming Languages conference in 1978 [7] ("HOPL-I") and the second one ("HOPL-II") in 1993 [8]. While these works are primarily devoted to tracing the origins and evolution of individual programming languages, the keynote address of HOPL-II, "Language Design as Design", by Frederick Brooks is relevant to the notion of ever-evolving languages. In this keynote, Brooks poses the question: "What have the existing high level languages contributed to software?" As he notes, the high level languages have first resulted in a "five-to-one productivity improvement" and secondly have resulted in a much higher level of software reliability by simplifying the syntax of statements, for after all "if you cannot say it, you cannot say it wrong". But most importantly, as Brooks points out, "the high level languages by their abstraction have given us ways of thinking and ways of talking to each other". Instead of talking to the machine, we begin to talk to one another; instead of spending time solving problems related to communication with the machine, we can spend time solving problems – our problems -- that lend themselves to programmatic solution. These thoughts and sentiments are to be found at the very genesis of language evolution, and they are echoed more recently as well.

In a 2002 Webcast interview on theserverside.com [9], Shawn Bayern, reference implementation lead for the JSTL, was asked to comment on the JSTL "expression language". The transcript of his response is telling:

> …[T]o a certain extent, all languages share a common bond and we are to a certain extent just replacing Java with a different language here. And we don't do that because we don't like Java, we do it instead because we don't think our users know Java necessarily. The advantage of an expression language is that users who use it don't have to understand Java types. They don't have to understand method invocation, exception handling, all of the different syntactical ways of producing an expression in Java, you know, all of the different Java productions because again, you have to know all of them if you're going to maintain pages that use them; you know, even if you've never seen the conditional operator or some feature of Java syntax that you might encounter, like a scriptlet. So the JSTL expression language hides all of that. It does the type conversions that are by and large appropriate and still safe to do in this environment so that you don't have to worry about whether you've got a string or a number as long as it's a parsable number, a simple number, you can pass it through. To give you an example all request [parameters] come into a JSP page as strings so if you want to pass this to a paging tag or something that let's you display boundaries of data, 'show me the first through the tenth', and you want to get that from a request parameter, you in Java have to convert it to a number and have to know how to do that. In JSTL you don't have [to]. You just say param.foo and you've got the number as a number [10].

The JSTL expression language hides much of the complexity of the underlying Java language in much the same way that 3GLs hide the complexities of the underlying machine code into which they compile. The principle of abstraction is at work here and seems to be one of the main motives fueling Bayern's efforts in creating the reference implementation of this language. In the end, the same work gets done – in this case the work consists of gleaning, then using, request parameters to a page – but without the complexity.

Bayern and Brooks would agree: Movement in the evolution of programming languages serves to make it easier to simply state, and to solve, the problems appropriate for exploration and resolution through the use of computing technologies.

Criticisms

There are at least three criticisms of the thesis here under consideration: (1) Proposals for what sort of languages should occupy the Fifth Generation designation was already made long ago; (2) Web-based tag libraries are not general purpose programming languages and so we should not seek to attribute an all-encompassing designation to what is a specialized domain of language; and tied to this (3) Web-based tag libraries are, in fact, too abstract to be generally useful.

It is true that the phrase "Fifth Generation Language" was once used to refer to two very different types of programming languages. First, it was used in the eighties and nineties to refer to "constraint" based languages like Prolog and Lisp, languages used in artificial intelligence research and in the construction of expert and knowledge-based systems. It, alternatively, was also used to describe the so-called "visual" programming languages, e.g., Visual Basic – languages that in their time were unique in relying heavily on graphical code generators. Nevertheless, it does not seem like the designation "Fifth Generation Language" has stuck to either of these concepts, and, if my thesis that generations of languages follow a path of abstraction away from machine code is taken seriously and construed as being the main principle behind the movement from generation to generation, nor should it.

The second criticism, that Web-based tag libraries are too specialized to deserve such a general designation as "Fifth Generation Languages" is more problematic. On the one hand it is easy to sympathize with the criticism that Web-based languages are only good for a particular domain of activity, Web-application development, and that whatever we end up calling Fifth Generation should apply to a much broader domain. In short, if we are to move to a new generation of language then that language or that set of languages must be general enough to warrant it. However, this criticism overlooks the fact that the technological and computing landscape in this, the early 21st century, has dramatically changed. At this point, even the notion of operating system is in flux. At this point, the Web itself has become the "cloud" into which data is dispersed, stored, and retrieved. Small chunks of executable code are distributed throughout this cloud, and individual applications are "mash-ups" of them, creating novel and unique molecules of functionality out of atomic modules and components. Web-application programming is no longer in a period of wait-and-see; it is here-

and-now. And Web-based tag libraries, while not universally adopted as of yet, promise to be its next generation of languages. They are, indeed, the general purpose languages of the Web.

The third criticism, if it is accepted as valid, is that Web-based tag libraries are in fact too general, i.e., they don't give you access or low-level control over such things as network protocols and database connectivity. Entire books have been written about JDBC, for example, illustrating the myriad connection attributes and fine-grained control over database connectivity one has in the Java world. However, the problem with this is that not only do you have the opportunity for such control with JDBC but you also *must* issue your connectivity commands using that fine-grained control if you are to accomplish anything. Now the question arises, how often will you actually need to use those low-level features? If the answer is "never" or "almost never", then perhaps it's best to abstract what you are doing to a higher level? Isn't this exactly what the designers of 3GLs sought when they moved away from machine code? This is precisely what the tag libraries have done for Web application programming. So criticizing Web-based tag libraries as being too abstract misses their point.

COMMENT AND CONCLUSION

The simplicity with which the Web-based tag library accomplishes its goals recalls, again, Frederick Brooks' trenchant comment about spare syntax: "[I]f you cannot say it, you cannot say it wrong". And this naturally has implications for developer productivity. It is a commonplace that developer productivity can be quantified by measuring lines of code produced over a given period of time. If a programming task can be accomplished in less lines of code, it stands to reason that productivity will increase. Build the kind of simplicity necessary to achieve this into the programming platform itself and productivity will increase, as Brooks noted, by orders of magnitude. The continuing work of Professor Lutz Prechelt at the Freie Universität Berlin bears this out. His Software Engineering research group there has devised the "Plat_Forms" project [11] -- "a competition in which top-class teams of three programmers compete to implement the same requirements for a web-based system within 30 hours, each team using a different technology platform (e.g. Java EE, NET, PHP, Perl, Python, or Ruby on Rails)." Further, "[i]ts purpose is to provide new insights into the real (rather than purported) pros, cons, and emergent properties of each platform." The first Plat_Forms contest was held January 25-6, 2007 in Nürnberg, Germany. Unfortunately, the three platforms tested, Java, PHP, and Perl, did not include tag libraries. Nevertheless, to briefly summarize the findings of this first competition, the languages requiring the smallest number of lines of code to accomplish the assigned tasks were Perl in first place, PHP in second, and Java in third. At first, it seems odd that syntax-heavy Perl would take the lead in this, and yet Perl is also infamous for having an Obfuscated Code Contest [12] in which the winning entries were notorious for completing the task in a very short string of head-scratchingly ponderous code: Perl is syntax heavy, yet it is far enough away from the machine that it packs a lot of functionality into each language construct, so much so that its terseness lends itself to easy (and hilarious) obfuscation. It is not surprising, therefore, that it was able to satisfy the requirements of the Plat_Forms competition in less lines of code than the other languages. That said, the hope is that the next Plat_Forms competition, to be held in 2008 or 2009, will include entries implemented in the tag libraries mentioned in this article. While the results of the first contest lean toward the conclusion that abstraction is the essence of simplicity when it comes to programming languages, only through empirical verification such as that offered by future Plat_Forms contests can such a judgment be validated.

If the argument that the generations of computer languages should be determined based upon the principle of abstraction, i.e., based upon how far from machine code they are in a functional and expressive sense, and how much closer to natural language constructs they are, then it follows that the conventional evolution from first to second to third to fourth generation languages is flawed. Specifically, the fourth generation seems out of line with the continuity of the previous evolution of languages. Arriving after the 3GLs were the scripting languages, which represent an abstraction away from the 3GLs toward simpler, more powerful natural language expressiveness and functionality. And at the end of this spectrum, at the current time, reside the new Web-based tag libraries, which are simpler and more powerful than the scripting languages. In order to better account for and to categorize these phenomena, it is better to consider scripting languages as the fourth generation of computer languages, and Web-based tag libraries as the fifth generation.

In the current IT climate, circa early 2008, there is movement afoot from all quarters to streamline systems languages such as Java and to abstract away their complexities without sacrificing practical power. In the Java world, frameworks such as Struts and Spring [13] are intended to simplify the design and construction of J2EE (Java 2 Enterprise Edition) Web applications. On the Java platform alone one can now write software in the Tcl, Python, and Ruby scripting languages using the Jacl, Jython, JRuby interpreters that are written in Java and run within a Java Virtual Machine (JVM). Moreover, with the advent of the so-called "Java scripting languages" [14], e.g., Groovy, BeanShell, JudoScript, language designers are implementing scripting languages on top of the Java platform that use Java syntax, albeit greatly simplified and distilled down to its most useful essentials. Just like with tag libraries, the principle here is the same: Abstract away the language complexity, move its expressiveness further away from the machine and closer to natural language, and the end result is a language that is simple, powerful, and even elegantly beautiful. It has been my goal in this paper to illustrate that throughout the history of computing the "generations" of languages have traditionally followed this path, and that this selfsame path points the way toward future generations.

This path seems to converge on a language, or family of languages, that is pared-down, simple, understandable, and elegant, yet capable of being used to write applications of extraordinary complexity and power. It seems to point toward a future in which the languages and frameworks we use fade into the background, becoming merely the manner in which we express the complex algorithms and applications that enable and facilitate the computing work of the world.

BIOGRAPHICAL INFORMATION

Mark Cyzyk is the Scholarly Communication Architect in the Library Digital Programs group of The Sheridan Libraries at Johns Hopkins University, Baltimore, Maryland, USA. He was, for many years, the Web Architect in the enterprise IT group at Johns Hopkins.

REFERENCES

[1] The seminal article on this topic is: J. K. Ousterhout. "Scripting: higher level programming for the 21st century." *IEEE Comput.*, vol. 31(3), pp. 23-30, March 1998.

[2] In this sense, tag libraries are "declarative" languages, not "imperative" languages, i.e., they *state* what they want the computer to do and don't therefore pay much attention to *how* it must implement a particular action. For more about the declarative/imperative distinction, see: M. Scott. *Programming language pragmatics*. San Francisco, CA: Morgan Kaufman, 2000, pp. 5-6.

[3] For a fascinating and important study of this, see: L. Prechelt. "An empirical comparison of seven programming languages." *IEEE Comput.*, vol. 33(10), pp. 23-29, October 2000.

[4] This code was taken, then modified, from one among the fine ASP tutorials posted to: Internet: www.asp101.com [March 4, 2008].

[5] M. S. Mahoney. "Software: The self-programming machine," in *From 0 to 1:An authoritative history of modern computing*. A. Akera & F. Nebeker, Eds. New York: Oxford University Press, 2002, p.95.

[6] J.E. Sammet. *Programming languages: History and fundamentals*. Englewood Cliffs, NJ: Prentice-Hall Inc, 1969.

[7] R.L. Wexelblat. *History of programming languages*. New York: Academic Press, 1981.

[8] T.J. Bergin & R.G. Gibson. *History of programming languages (HOPL-II)*. New York; Reading, Mass.: ACM Press; Addison-Wesley, 1996.

[9] Internet: www.theserverside.com/talks/videos/ShawnBayern/interview.tss?bandwidth=dsl [March 4, 2008].

[10] Internet: www.theserverside.com/talks/videos/ShawnBayern/text/txt09.html [March 4, 2008].

[11] "Plat_Forms: The web development platform comparison." Internet: www.plat-forms.org [March 4, 2008].

[12] J. Orwant, Ed. "Best of the Perl Journal: Games, Diversions & Perl Culture." Internet: www.oreilly.com/catalog/tpj3/chapter/ch43.pdf [March 4, 2008].

[13] "Java's Ultimate Skeptic," *Information Week*, p. 62, December 17/24, 2007. This brief article cites Spring architect Rod Johnson as "Java's Ultimate Skeptic." This because he "began to question the fundamental tenets of the language" and began to construct his own framework, Spring, to deal with the complexities of J2EE development.

[14] T. Grall. "Scripting Languages Ease Development and Administration." *Java Developer's Journal*, vol. 10(11), November 2005, pp.10-12.

Received: January 23, 2008 Revised: March 05, 2008 Accepted: March 06, 2008