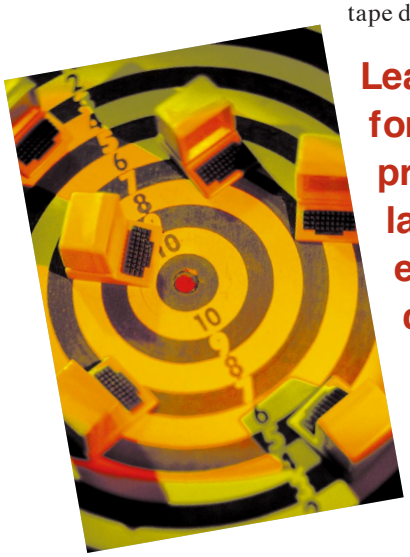
How Programming Languages Evolve

Tom Jepsen

Once upon a time, life was simple. Large monolithic computers, usually painted blue, ran single-threaded batch programs under the watchful eyes of an operator and a system programmer. A card reader served as the input device, a tape drive provided storage,



Learning what forces drive programming-language evolution can help you pick one for your project.

and a line printer processed output. Programmers wrote business applications in Cobol, scientific applications in Fortran. In either case, they worked out the program logic on paper first, used a keypunch to produce punched cards, then ran the resulting deck through the card reader. After a few debugging sessions, they received their computed results on sheets of fanfold paper. If a program required documentation, programmers produced it on a manual typewriter.

Today, a few lumbering programming languages cling to life while hordes of newer types struggle to replace them. These new species include a confusing swirl of languages that are

- compiled,
- interpreted,
- Web based,
- scripting and modeling capable,
- object oriented,
- graphically based,
- text-processing based, or
- founded on artificial intelligence routines.

In addition, all kinds of specialized languages address the challenges of developing specific applications. To an outside observer, this proliferation might seem strange. Why haven't computer scientists and IT professionals—people generally scientific and rational almost to a fault—focused on creating a few robust and adaptable problem-solving languages that could succeed in any computing environment?

ECOLOGICAL NICHES

Quite simply, the rapid spread of computerization into all phases of modern life—and the diversity of application and problem domains this trend has created—require a wide range of problem-solving tools.

Programming languages have evolved to provide such tools and, like evolution in nature, have generated numerous mutations in the process, some successful and some not. A few forces drive this evolutionary process; understanding them and which languages evolved as an answer to these forces may help you select a language for your development project.

Send in the clones

Much of software programming consists of reinventing the wheel with a slightly different color scheme, turning radius, and spoke orientation. This realization launched programmers on a quest to develop techniques for cloning and recycling code used on previous projects. Structured programming led to code modularization, which in time led to object-oriented programming and component-based development. The quest for reusable software also contributed to the development of architecture- and platform-independent languages.

The incredible shrinking computer

With each succeeding generation, computers have housed the same processing power in smaller packages: First, mainframes shrank to minicomputers, then minicomputers dwindled to desktops. Now ever smaller computers find their way into handheld personal digital assistants, clothing, and jewelry. Computers have become ubiquitous. Almost all technological artifacts now contain a stored program of some sort. Such small computers demand economical languages that leave a comparably small footprint.

Have code, will travel

Manufacturers of mainframes and even early PCs designed their products for standalone

1520-9202/99/$10.00 © 1999 IEEE

operation. The development of local area networks, client-server architecture, and the Internet have vastly altered the nature and scope of computing. Software's physical location is no longer an issue: Components can execute locally or travel to another location to execute remotely. Object brokers can be used to set up temporary "relationships" between units of software function and requesting clients in a distributed object environment. Operation in this environment demands mobile, platform-independent code.

SELECTING FOR SPEED AND FLEXIBILITY

According to Perl creator Larry Wall, "Real languages evolve, they don't revolve. Truly revolutionary computer languages tend not to catch on" ("An Ongoing Revolution," *Computer*, May 1999, pp. 48-49). Companies with large investments in legacy software show an understandable reluctance to experiment with novel approaches to programming, contributing to the tendency of programming languages to stick around until they become "dinosaurs." Take a snapshot of any moment in the history of programming-language development, and you will see the same picture. Even as these lumbering dinosaurs take gigantic bites out of the market share jungle, smaller, faster-moving mammals scurry about, seeking an overlooked niche they can occupy while evolving into something greater. The history of programming language evolution chronicles the slow but steady triumph of these compact speedsters over lethargic giants. Managers and programmers who are able to spot the evolutionary winners early on are destined to be successful in a competitive environment.

Early mammals

Created by Dennis Ritchie in the 1970s, C became one of the first speedy mammals to emerge from the programming jungle. In their preface to *The C Programming Language*'s second edition (Prentice Hall, Englewood Cliffs, N.J., 1988), Ritchie and coauthor Brian Kernighan acknowledged that they designed C to be deliberately minimalist, noting that "C is not a big language, and it is not well served by a big book." The two programmers developed C to meet the twin and sometimes contradictory goals of architecture independence and a high degree of programmer control over machine operations. Thus, while C gave programmers the ability to explicitly allocate and deallocate memory, it also made the length of an integer depend on the underlying platform.

Ken Thompson developed the Unix operating system in tandem with, and largely based upon, C. Thompson originally intended for Unix to be platform independent; one of its distinguishing features is its generic open/close/read/write input/output abstractions. Unix, however, quickly developed dependencies on the underlying hardware. The need to port Unix to different platforms contributed to the proliferation of Unix flavors and to the resultant inability of any one version to capture a large user market share.

OO invades the desktop

The quest for reusability spawned the notion of the object: a small, reusable unit of software that encapsulates behavior in terms of data and the methods that operate upon it. An early OO species, Smalltalk, proved well-suited to modeling and rapid prototyping. A small, interpreted language with weak typing, Smalltalk was one of the first languages to use automatic garbage collection, a feature in which allocated memory that is no longer being used is automatically released, without requiring an explicit call by the programmer. Automatic garbage collection makes memory management easier and less error-prone. Ultimately, though, Smalltalk never evolved into a language for large-scale software development because, as an interpreted language, it lacked runtime efficiency. It also lacked commonly required functions, such as the ability to convert numbers to strings.

The development of C++ by Bjarne Stroustrup in the late 1980s overcame many of Smalltalk's shortcomings. A compiled language like its ancestor C, C++ provided good runtime performance. It also provided strong typing, which resulted in better error checking and optimization. These qualities helped C++ move OO into the programming mainstream.

Yet C++ had its own drawbacks. Complex and difficult to learn, it suffered from unstructured class libraries that required programmers to spend time looking things up. C++ also inherited C's platform dependence.

Despite their imperfections, C and C++ formed the backbone for much of the systems programming that underpinned the personal computer revolution of the early 1990s, including the development of Microsoft's DOS and Windows operating systems. Another language filled the personal computer niche when IBM

What Interests You?

As the editor for programming languages, I'd like to bring you information from industry leaders and visionaries who will address the scope and direction of change in software. I'd like them to talk about the evolution of object-oriented computing, the next-generation user interface, and the open source movement. Please contact me at tom.jepsen@fnc.fujitsu.com with comments on these ideas or suggestions for other topics.

Tom Jepsen is a systems engineer in switch planning at Fujitsu Network Communications.

How Typing Handles Program Variables

Typing refers to the assignment to a category of each declared variable in a program. Such categories—which include integer, float, string, or class—determine how the variable will be understood and used by the software. Typing has advantages for the programmer who writes the source code and the compiler or interpreter that generates the object code. For the programmer, having a standard set of data types provides added readability, reliability, and maintainability. Knowing a variable's type enables a compiler or interpreter to allocate the right amount of storage for the variable and to process its contents correctly.

Types may be implicit or may be declared explicitly. Fortran IV provided implicit typing by default—if you declared a variable name that began with I, J, K, L, M, or N, the language automatically treated the variable as a fixed-point integer; Fortran IV assumed all other variables to be floating point. Java, on the other hand, does not allow default types; all variable declarations must specify a type.

Developers classify programming languages as *weakly typed* or *strongly typed*. Weakly typed languages, like Smalltalk and Tcl, do not require variables to have a type declaration and place no restrictions on how variables can be used. Weakly typed languages are generally interpreted languages, with a sparse syntax and good rapid-prototyping capabilities. Strongly typed languages, on the other hand, like C, C++, and Java, require explicit type declarations for all variables. Normally, developers compile such languages and the compiler does extensive type checking at compile time to ensure data-type compatibility. If the compiler detects an incompatibility, it generates a compilation error message and terminates the compile operation.

Adding two and two in strongly typed C requires several lines of code:

```
int a = 2;
int b = 2;
int c;
main() {
c = a + b;
printf("%d\n", c);
}
```

The same function in weakly typed Tcl can be written as

```
set a 2
set b 2
expr $a + $b
```

However, in some cases, weak typing can lead to ambiguity. In Smalltalk, all variables can be considered to be of type "object" since all variables are objects by default. But what does it mean to say that two objects are equal or identical? For example, testing the integer 5 for identity with itself

```
5 == 5
```

will evaluate as true, since the Smalltalk interpreter recognizes the small integer 5 as an identity object. However, testing the fraction 1/5 for identity with itself

```
(1/5) == (1/5)
```

will evaluate as false, since the program evaluates the expression at runtime, and creates and compares two different objects.

packaged Basic, a simple interpreted language, with its original PC. Basic gave many new PC owners their first exposure to programming. It would later evolve into Visual Basic, a powerful tool for developing graphically based Windows applications.

Branching into text

Another evolutionary path began with the use of computers for text processing. Computer programs needed user documentation; it made sense to develop the documentation on the computer that ran the program. Script, a text processing language developed at Canada's University of Waterloo in the 1970s, provided commands for formatting text in written documents and quickly came into general use on mainframe computers.

Researchers at IBM created a text-processing macrolanguage based on Script, called the Generalized Markup Language. GML let users create a document by simply assembling the necessary building blocks of headings, paragraphs, and formats. Some developers noticed that text processing using GML resembled object-oriented programming: You could create a generic functional block, such as a paragraph, that could be customized for a specific document by specifying an instance with a specific font, page layout, and so forth.

This comparison resulted in the development of the Standard Generalized Markup Language. SGML includes the concept of a document type definition, which specifies a set of elements and the tags used to define each document element. DTD, the specification for hypertext markup language (HTML), would play a leading role in a truly revolutionary development—the growth of the Internet and the World Wide Web.

Weaving a worldwide web

Although the Internet had its origins in the Arpanet work sponsored by the US Department of Defense's Advanced Research Project Agency in the 1960s, it really took off when large-scale networking became feasible through academic institutions' widespread use of Unix in the early 1980s. However, the original text-based Internet of the 1970s and 1980s formed an arcane, hackers-only world that few outside computer science departments could enter. This situation began to change in 1989 when Tim Berners-Lee of CERN, envisioning a shared workspace for collaborative scientific work, developed the Hypertext Transport Protocol (HTTP) and HTML, thereby laying the groundwork for the World Wide Web. However, the real revolution began in 1993, when Marc Andreessen, a young programmer at the National Center for Supercomputing Applications, created Mosaic, a graphically based client for Web browsing. Mosaic opened the online community to the general public and initiated the fast-paced growth in Web-based applications that continues today.

While early Web pages were limited to static display of information, Web applications quickly became more interactive and required complex server-side functions, such as database lookups and dynamic updating of HTML content by means of common gateway interface (CGI) scripts. This requirement in turn created the need for scripting software that could provide the "glue" between HTML user requests and server-based applications written in conventional programming languages.

Script doctoring

The concept of scripting languages is actually quite old: IBM developed Job Control Language in the 1960s as a scripting language for mainframe computers. Programmers used JCL to initiate execution of applications written in other languages. In the 1990s, Visual Basic, though originally intended as a system programming language, came into widespread use as a scripting language for developing Windows applications using components written in C or C++.

Java's added features, including its downloadability and ability to provide executable content, have led to its instant popularity.

Perl, developed by Wall in the early 1990s, has quickly become the language of choice for providing Web servers with CGI and general administrative functions. Interpreted, weakly typed, and platform independent, Perl's scripts run equally well on the various flavors of Unix, Windows NT, and MacOS. The language, which can still be downloaded and used free of charge, also became one of the first widely available open source languages.

First released in 1990, John K. Ousterhout's Tool Command Language—another "glue" language—enjoys a large following. Ousterhout designed Tcl (pronounced "tickle") as a scripting language for integrating applications written in other languages. Open source and essentially typeless, Tcl lets users develop graphical applications by stringing together user interface "controls" similar to Visual Basic.

Ultimate adaptability

Scripting languages solved the problem of providing interactivity between Web users and server-resident applications. This solution does not work well, however, if all clients try to access the same application at once. A more scalable way of providing interactivity would be to offload the server application by distributing parts of its functionality. One approach to achieving such offloading involves downloading software to requesting clients for local execution. With Java's introduction in 1995, this method became a reality.

Developed by James Gosling and others at Sun Microsystems, Java and its applet-based animations soon danced across Web pages around the world, eating up client PC processor cycles rather than the CPU time of the servers from which users downloaded it. Java's developers, however, envisioned it as capable of far broader application than serving as a tool for decorating Web pages. They designed Java as a general-purpose programming language that would provide reusability, a small footprint, and architecture independence.

An advanced hybrid, Java combines the strengths of many predecessors. It retains the strong typing and object orientation of C++, but eliminates complexities like pointers. Java shares Smalltalk's capacity for automated memory allocation, Perl's platform independence, and Visual Basic's excellent GUI-development capabilities. But Java's added features, including its downloadability and ability to provide executable content, have led to its instant popularity, creating possibilities for truly distributed computing. Currently, Java and the JavaScript scripting language compete with Microsoft's ActiveX controls to become the industry standard

IT Professional

ADVERTISER | PRODUCT INDEX

NOVEMBER | DECEMBER 1999

for providing Web-based executable content.

FUTURE GENERATIONS

Although Java appears to hold the current evolutionary crown, I still wonder where programming-language evolution will lead next. Object-oriented software continues evolving into component software, with standardized interfaces and functions. To quote Bertrand Meyer ("On to Components," *Computer*, Jan. 1999, pp. 139-140), "A true component must be usable by software developers who build new systems not foreseen by the component's author." Add downloadability and you have distributed components that let you create network architectures without worrying about where the software actually resides. Put additional intelligence into the mix, and you have mobile agents capable of moving through the network in a heuristic fashion and interacting intelligently with their execution environments. Web-based languages continue evolving new capabilities as well: The Extensible Markup Language (XML) promises to extend HTML's capabilities by letting users create customized document component types and by permitting links to multiple sources. Combining XML and scripting languages will likely result in even more dynamic behavior.

Will programming languages continue to proliferate, or do languages like Java represent a trend toward convergence? Will truly user-friendly languages develop that will enable nontechnical people to create programs? Will the open source movement make software development more democratic and thus more responsive to the diverse needs of various users? Will the movement produce more reliable software? I plan to explore these and similar issues in future articles. ■