# Selecting a Programming Language for Your Project

**David Naiditch**
*Raytheon Systems Company*

## INTRODUCTION

In 1997, the mandate was dropped that the Ada language had to be used in all DoD projects. Since then, software managers have had more freedom to choose which programming language to use on their DoD projects. With this new freedom has come new responsibilities. Too often, discussions about what language to select break down into emotional appeals for one's favorite language. Instead, a fair and rational process should be established that bases the choice of a programming language on issues related to customer needs and business goals. In particular, for large projects, a language should be selected according to objective evaluation criteria, a system of priorities, and an overall scoring algorithm. This selection process should be done by those having a working knowledge of the languages under consideration. The winning language that emerges from such a process might not be the language initially considered as the obvious best choice. For projects that are too small to afford the time and effort needed to evaluate different languages, the choice of language might be made at the product line level or higher.

For most new DoD software projects that need a general purpose high-level language, Ada, C, or C++ are the main contenders, with increasing attention paid to the new language celebrity, Java. (In this paper, "Ada" refers to both Ada 83 and Ada 95.) Furthermore, multilanguage systems seems to be getting increasingly popular. As a result, companies such as Rational, Aonix, and Green Hills offer integrated multilanguage programming environments. (In response to this trend, Ada 95 has added several new pragmas [compiler directives] and special packages [Interface and its child packages] that enable Ada 95 code to easily interface with other languages.)

As Table 1 shows, there are many important differences between Ada 95, Ada 83, C++, C, and Java. The comments in the right column of the table provide some rationale for the rankings. A full discussion of each category in Table 1, on next page, is beyond the scope of this brief paper, but some issues are discussed subsequently when I cover common misconceptions.

Although each programming language has its own strengths and weaknesses, the reason for choosing a particular language may ultimately be based on factors having nothing to do with the technical merits of the language itself. (This is yet another reason that the Ada mandate was dropped.) Such factors may include the availability of compilers for the host/target, the maturity and efficiency of available compilers, the availability of programmers who already know the language, whether the language can easily interface with existing systems such as graphical user interfaces or data bases, the existence of legacy software written in a specific language, or how well the language fits in with adopted commercial-off-the-shelf (COTS) products.

A few common misconceptions that may affect the choice of a programming language are listed below.

## MISCONCEPTION 1: THE GOVERNMENT DROPPED THE ADA MANDATE BECAUSE ADA IS A DYING LANGUAGE

When, in 1997, the Honorable Emmett Paige, Jr., Assistant Secretary of Defense for Command, Control, Communications and Intelligence, dropped the Ada mandate, he made it very clear that this action should not be seen as the government's lack of commitment to Ada. He felt that the mandate became a "contentious point of

Author's Current Address:
Raytheon Systems, Company, RE Building, RO1 M/S A521, 2000 East Imperial Highway, El Segundo, CA 90245, USA.

Based on a presentation at DASC '98.

**Table 1. Language Comparisons**

| Feature | Ada95 | Ada83 | C++ | C | Java | Comments |
|---|---|---|---|---|---|---|
| Built-in support for concurrency (multithreading) | Excellent | Very Good | None | None | Good | For C or C++, concurrent programming requires calls to nonportable and low-level operating system (OS) services. |
| Full support of object-oriented programming | Yes | No | Yes | No | Yes | Ada 83 has most, but not all object-oriented features. |
| Full support for procedural-based programming | Yes | Yes | Yes | Yes | No | To use Java, object-oriented programming must be fully embraced. |
| Support of modern software engineering principles | Excellent | Excellent | Average | Poor | Good | Principles include strong typing, information hiding, localization, modularity. Ada is unique in strong type checking of integer types and array index types. |
| Completeness of language features | Excellent | Good | Very Good | Poor | Poor | Java lacks many common language features, such as enumeration types and records/structures. Ada 83 lacks pointers to subprograms and stack-allocated objects and certain OOP features. C++ lacks concurrency features. |
| Portability | Very Good | Very Good | Average | Good | Excellent | Java's portability derives mostly from the Java virtual machine, not from the language itself. C++ implementations still do not fully conform to the recently introduced ISO standard. |
| Life cycle costs | Excellent | Excellent | Poor | Poor | Good | Historic data show that, line per line, Ada code is about half the cost of C or C++ code (see reference to Zeigler's article). |
| Safety and reliability | Excellent | Excellent | Poor | Poor | Very Good | Ada catches many bugs at compile time and then others at runtime. Ada is the most strongly typed language and guards against dangling pointers. |
| Ease of reuse | Excellent | Excellent | Excellent | Poor | Good | Neither Java nor C have generics/templates. C's macro expander is a poor substitute. |
| Support for inter-language communication | Very Good | Poor | None | None | None | Ada 95 provides several pragmas (compiler directives) and packages for this purpose. |
| Language support | Good | Fair | Good | Very Good | Excellent | Many Ada vendors have transitioned to Ada 95 and consider Ada 83 to be obsolete. Ada host and target compilers are readily available for mainstream microprocessors, but may be unavailable for special microprocessors. |

resentment," and that Ada was fully capable of standing on its engineering merits.

In fact, Ada is very unlikely to go away in the foreseeable future. The DoD has a tremendous investment in Ada. Over 50 million lines of Ada code have been developed for warfighting systems. Many of these systems will be used well into the next millennium. Furthermore, Ada will continue to be chosen for its reliability and lower development/maintenance cost, both of which are so important for large, long-lived software systems (see Misconception 11).

## MISCONCEPTION 2: C IS THE MOST EFFICIENT LANGUAGE SINCE IT IS CLOSER TO ASSEMBLY CODE

C is not necessarily the most efficient language. When runtime checks are turned off, Ada code may run about as fast as C. (Standard C does not perform runtime checks.) Java, however, usually runs considerably slower because it makes extensive use of dynamic (heap allocated) data structures and has automatic garbage collection, and because the generated bytecode is typically interpreted. (The last problem can be mitigated if a "just-in-time" [JIT] compiler, or better yet, a native Java compiler, is available.) Of course, efficiency can significantly vary from one implementation of a language to another. Furthermore, different compilers are optimized for different language features.

## MISCONCEPTION 3: VERY TERSE SOURCE CODE RESULTS IN VERY EFFICIENT OBJECT CODE

The only guarantee is that very terse code results in unreadable and, therefore, unmaintainable code. Do not assume, for example, that this C code

```
if ((A = B) == C) C += 2;
```

executes any faster than

```
A = B;
if (A == C) C = C + 2;
```

Interestingly, Java's syntax is a somewhat cleaned-up version of C++'s syntax. For instance, Java eliminates the possibility of confusing an assignment with a test for equality in a conditional. Thus, Java rejects this code

```
if (A = B)
```

However, Java embraces C's error-prone syntax for the switch (case) statement, where a "break" can be accidentally omitted.

**MISCONCEPTION 4:**
**EASE OF CODING IS A MAJOR ADVANTAGE**

Ease of coding is not a major advantage compared with ease of reading. For complex code that must be maintained over many years, maintenance costs typically run 70% to 90% of the software life cycle costs. Because one must be able to read code in order to maintain it, the extra effort taken in developing readable code amply pays off. And don't forget that the annoyance of excess key strokes is no excuse for writing unreadable code, not even for two-finger typists.

**MISCONCEPTION 5:**
**SOFTWARE SHOULD BE EASY TO COMPILE**

Software should not necessarily be easy to compile. As many bugs as possible should be caught at compile time, since the sooner a bug is caught, the easier and cheaper it is to fix. Ada, for example, catches many errors at compile time (for example, calling subprograms with the wrong type or number of parameters) than other languages catch at runtime, or not at all. As a result, Ada code is harder to compile than, for example, C code, but once compiled, many potential errors have been eliminated.

**MISCONCEPTION 6:**
**JAVA IS A GREAT LANGUAGE BECAUSE THE SAME CODE CAN RUN ON WINDOWS 95, SOLARIS, UNIX, MACINTOSH, OR JUST ABOUT ANY PLATFORM**

Java's platform independence is wonderful (although a bit overstated). However, this independence derives not so much from the Java language itself, but from the ability of the Java compiler to generate Java bytecode that can be interpreted on any machine with a Java-enabled web browser such as Netscape or Internet Explorer. Note that Java bytecode does not have to be generated by a Java compiler. For instance, Intermetrics and Aonix offer Ada 95 compilers that generate Java bytecode.

Although Java's portability does not have that much to do with the language itself, Java does address portability through predefined numeric types with specified sizes and formats. Thus, type "int" is always a 32-bit signed integer type, whereas in C and C++, int can be of any size. If any of Java's numeric types are not supported by the target architecture, then that type must be implemented in software, which could significantly degrade performance (see Misperception 2). In Ada, a programmer can select a numeric type by range and/or precision rather than only by name, which guarantees that the numeric type has the range and precision that is required.

**MISCONCEPTION 7:**
**USING OPERATING SYSTEM (OS) SERVICES FOR CONCURRENT PROGRAMMING IS JUST AS GOOD (IF NOT BETTER THAN) USING SPECIAL LANGUAGE CONSTRUCTS**

Unlike Ada and Java, C and C++ do not provide constructs for writing concurrent code (code that can do more than one thing at once). To write concurrent software, C and C++ programmers often rely on OS services such as semaphores that are nonportable, low level, error prone, and difficult to debug. For example, with semaphores, there is a risk of locking (seizing) an item and then forgetting to unlock (release) it when one is finished with it so that other tasks can access the item. In contrast, Ada 95 allows protected operations to be defined that automatically provide mutually exclusive access to a protected item. This feature eliminates the risk of having an unbalanced lock/unlock. (For hard real-time programs that cannot even afford the overhead of protected types, Ada 95 offers portable semaphore services.)

**MISCONCEPTION 8:**
**PROCEDURAL-BASED PROGRAMMING HAS BEEN REPLACED BY OBJECT-ORIENTED PROGRAMMING**

Although the programming world has largely gone object-oriented, applications do exist (for example, embedded real-time or scientific applications) where procedural-based programming seems more natural. In Ada 95, the language features that support object-oriented programming are distinct and can be used separately. Therefore, Ada 95 programmers are free to select those object-oriented features they want without being forced into fully embracing the object-oriented methodology. In contrast, Java only supports object-oriented programming. Java cannot be used for procedural-based programming. (In Java, even exceptions and threads are defined in object-oriented terms.) C++ is like Ada 95 in that it does not force programmers to embrace object-oriented programming. However, object-oriented features of C++ are not as distinct as those of Ada 95. For instance, data

encapsulation, which is needed for creating abstract data types (called private types in Ada), cannot be performed without introducing a class, which is a central feature of object-oriented programming.

**MISCONCEPTION 9:**
**JAVA IS SMALL, SIMPLE, AND EASY-TO-LEARN**

Even though Java is a small and simple language, it has some subtle features and includes a huge library of over 150 classes and interfaces. It takes considerable effort to determine which classes to use and how best to use them. Programmers accustomed to large languages such as Ada and C++ often find that Java is too small. For example, Java is missing local constants, method parameter modes, enumeration types, records/structures, operator overloading, variant records/unions, explicit pointers, fixed-point types, unsigned integer types, range constraints, the goto statement, and generics/templates. Some of these features were undoubtedly omitted in the interest of simplicity, portability, or security, but not all of these omissions are easy to justify.

In general, for small languages to be effectively used, programmers often need to write code that simulates a missing language feature, or to rely on tools or practices that are external to the language itself. For instance, Ada defines a basic library manager that prevents an executable module from ever being built from out-of-date components. Other languages obtain this capability through the use of a Make utility and the careful use of naming conventions. However, as a result, portability is lost and complexity increased, since the proper use of such languages depends on nonstandardized utilities and practices that exist outside the language. Thus language complexity should not be considered in isolation, but as part of the software engineering environment.

**MISCONCEPTION 10:**
**COMPUTER LANGUAGES SHOULD GIVE PROGRAMMERS THE FREEDOM TO DO WHAT THEY WANT TO DO**

This libertarian view is also held by those who resent the government forcing them to wear a motorcycle helmet. Without a doubt, Ada, and to some extent, Java, are "paternalistic" languages that attempt to protect programmers from themselves and other programmers. In a world where all drivers and programmers are perfectly rational and sober, such paternalism may seem unnecessary. However, on the freeways populated by maniacs, and in the real software development world where mind-bogglingly complex programs are being maintained over many years by gaggles of programmers, such optimism is unwise.

Consider, for example, the use of pointers. In C and C++, pointers can point willy-nilly to any address. Not surprisingly, the misuse of pointers is responsible for the majority of C and C++ programming errors. Often pointers end up pointing to the wrong data or to items that no longer exist. Java takes a Draconian approach to this problem of pointer abuse by not allowing programmers to explicitly declare pointers! In Ada 95, pointers are flexible, but also quite safe. Pointers can point to heap allocated objects (as in Ada 83), to stack allocated objects, and to sub-programs. Checks are automatically made (mostly at compile time) to guarantee that (in the absence of "unchecked programming") a pointer does not point to an object of the wrong type, or point to a stack allocated object or sub-program that no longer exists.

**MISCONCEPTION 11:**
**IN TERMS OF LIFE CYCLE COSTS, IT DOES NOT MATTER WHAT PROGRAMMING LANGUAGE IS CHOSEN**

Various studies have shown that the choice of programming language does impact life cycle costs. Such claims, however, are difficult to substantiate. Part of the difficulty is that large projects are never done in parallel with different languages, but with all other factors held constant. Perhaps the study that best approaches this ideal was done by Dr. Stephen Zeigler of Verdix (now Rational Software Corporation). Dr. Zeigler based his study on records kept at Verdix documenting all software changes. Although these records were not intended to be used to compare the development costs of Ada and C, they turned out to be very useful because few factors varied besides the language being used: The same 60 programmers were developing code in both Ada and C, the same work environment was used, the same debugging tools, same editors, same testing tools, and the same design methodology. Most of these programmers had masters degrees in computer science, and the more experienced programmers tended to work more in C. When first hired, 75% of the programmers knew C, while only 25% knew Ada. Despite the bias in C's favor, Dr. Zeigler showed that the cost of coding in Ada is about half the cost of coding in C. This difference in cost resulted because code written in Ada contained 70% less bugs discovered before product delivery and 90% less bugs discovered after product delivery. (As discussed under Misconception 5, the later a bug is caught, the more expensive it is to fix.) Dr. Zeigler expects the cost of C to be even more pronounced in other organizations, because Verdix had the advantage of understanding common C problems and enforcing thorough software testing procedures. Interestingly, C++ bugs were running even higher than C bugs. Dr. Zeigler's paper, "Comparing Development Costs of C and Ada," can be obtained at http://sw-eng.falls-church.va.us/AdaIC/docs/reports. ▣