Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

Choosing a Programming Language

Diomidis Spinellis

A language that doesn't have everything is actually easier to program in than some that do.
—Dennis M. Ritchie

Computer languages fascinate me. Like a living person, each one has its own history, personality, interests, and quirks. Once you've learned one, you can use it again after years of neglect, and it's like reconnecting with an old friend: you can continue discussions from where you left off years before. For a task I recently faced, I adopted a language I hadn't used for 15 years, and I felt enlightened.

I don't think there's one language suitable for all tasks, and probably there won't ever be one. In a typical workweek, I seldom program in fewer than three different languages. The most difficult question I face when starting a new project is what language to use. Factors I balance when choosing a programming language are programmer productivity, maintainability, efficiency, portability, tool support, and software and hardware interfaces.

Hard choices

Often, one of these factors is decisive and leaves little room for choice. If you have to squeeze your interrupt-driven code into a microcontroller's 1,024 bytes of memory, assembly language or maybe C is the only game in town. If you're going to interface with a Java-based application server, then you write in Java. Sometimes tradition plays an important role. Systems code, like operating systems, device drivers, and utility programs, is typically written in C. Following this tradition means that the code will mesh well with its surrounding environment and won't impose on it onerous requirements for libraries and runtime environments.

At other times, the choice of programming language is a fine balancing act. I find the power of C++ and its standard template library amazing: the combination provides me with extreme efficiency and expressiveness—at a price. The language is large and complex; after 15 years of C++ programming, I'm still often puzzled by the compiler's error messages, and I routinely program with a couple of reference books by my side. Time spent looking up an incantation is time not spent programming.

Modern object-oriented languages such as Java and C# are more orthogonal and hide fewer surprises for the programmer, although the inevitable accumulation of features makes this statement less true with every new version of each language. It looks as if Lehman's law of software evolution ("as a program is evolved its complexity increases") haunts us on every front. On the other hand, sometimes you just can't afford Java's memory space overheads. I recently wrote a program that manipulated half a billion objects. Its C++ implementation required 3 Gbytes of real memory to run. A Java implementation would easily need that amount of memory just to store the objects' housekeeping data. I couldn't afford the additional memory space, and I'm sure even our more gener-

 0740-7459/06/$20.00 © 2006 IEEE

ously funded CERN colleagues feel the same way when facing the 1 petabyte per second data stream coming from their large hadron collider experiment.

However, the situations I described are outliers. In many more cases, I find myself choosing a programming language on the basis of its surrounding ecosystem. If I'm targeting a Windows audience, the default availability of the .NET framework on all modern Windows systems makes the platform attractive. Conversely, if the application must ever run on any other system, using the .NET framework will make porting it a nightmare. Third-party libraries also play an important role here.

Nowadays, we often build applications by gluing together many libraries. I recently calculated that, on average, each of the 20,000 applications ported to the FreeBSD system depends on 1.7 third-party libraries not available on the system's default installation; one application depends on 38 different libraries. So, for example, if your application requires support for 3D rendering, Bluetooth communications, the creation of PDF documents, an interface to a particular relational database management system, and public key cryptography, you might find that these facilities are available only for a particular language.

Soft choices

When efficiency, portability, and library availability don't force a language on me, the next decisive factor is programmer productivity.

Interestingly, I've found that the same language features can promote or reduce productivity, depending on the work's scope. For small tool-type programs, I prefer a language that sustains programmer abuse without complaint. When I want to put together a program or a one-line command in a hurry, I appreciate that Perl and the Unix shell scripting facilities don't require me to declare types and split my code into functions and modules. Other programmers use Python and Ruby in the same way.

However, for programs that will grow large, be maintained by a team, or be used where errors matter a lot, I want a language that enforces programming discipline. One feature I particularly appreciate is strict static typing. Type errors that the compiler catches are bugs my users won't face. Language support for splitting programs into modules and hiding implementation details is also important. If the language (or the culture of developing in that language) enforces these development traits, so much the better. So, even though you can write well-structured, 100,000-line programs in both Perl and Java, the discipline required to get this right in Perl is an order of magnitude higher than that required in Java, where even rookie programmers routinely split their code into classes and packages.

I also pay attention to a language's supporting environment. Nowadays, a programmer's productivity in a given language often depends on using an integrated development environment. You really wouldn't want to approach some tasks, such as developing a program's GUI layout, without an appropriate IDE. Some colleagues have become attached to a particular IDE in the same way I'm clinically dependant on the vi editor. So, choosing a language often involves selecting one that a particular IDE supports.

Declarative choices

Sometimes, a program's application domain will favor a specific language's expressive style. The three approaches here involve using an existing domain-specific language (DSL), building a new one, or adopting a general-purpose declarative language.

Adopting an existing DSL is often a no-brainer: if you want to get some figures from a database, you might write SQL queries; if you want to convert an XML document into a report, you should try out XSLT. Building a special-purpose language might sound daunting, but it's not that hard if you take the right shortcuts. And, it can be a tremendous productivity booster. Fifteen years ago, I designed a simple line-oriented DSL to specify the parameters of a CAD system's objects. Instead of designing an input window layout for each group, you simply specify declaratively what the user should see and manipulate. So, the system's initial 150 parameters have effortlessly swelled over the years to 2,400, surviving a port to a different GUI platform intact.

When I recently set out to design a way to specify complex financial instruments, I first designed a DSL. However, the more I worked on the problem, the more I realized that declarative languages such as Prolog, Lisp, ML, and Haskell already had many of the features I wanted—list and tree manipulation, for example. After expressing a small subset of the problem in several of these languages, I singled out Haskell, a language I had to write a compiler for as an undergraduate student. It seemed to offer a concise way to express everything I wanted and a no-frills but remarkably effective development environment.

My biggest surprise came when I started testing the code I wrote. Most programs worked correctly the first time on. I attribute this to three factors. Haskell's strong typing filtered out most errors when I compiled my code. Furthermore, the language's powerful abstractions let me concisely express what I wanted, limiting the scope for errors (research has shown that the errors in a program are roughly proportional to its size). Finally, as a pure functional language, Haskell doesn't let expressions have side effects, and this forced me to split my program into many simple, easy-to-verify functions.

Over the years, many friends and books have prompted me to evaluate the use of a functional language for implementing domain-specific functionality. As I continue to add Haskell functions to my program, I can see how choosing the appropriate programming language can make or break a project.

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.