Multilanguage Programming

Steve Vinoski • *Verivue*

Have you ever worked on an integration project with a developer who possesses seemingly limitless knowledge, wisdom, and experience with the various systems and techniques required? We've all heard of — and might even know — individuals from various professions who are considered to be exceptional at what they do. Whether they're athletes, actors, mechanics, or software developers who focus on integration, such people all possess *extensive vocabularies*. Here, the term "vocabulary" refers not to spoken or written words, but more abstractly to the tools, tactics, and techniques pertinent to each profession. Top basketball players know multiple ways to help their teams with scoring, passing, and defense, and they can adapt their games as needed. Virtuoso musicians are skilled at multiple instruments, styles, and techniques. Seasoned integration developers tend to be knowledgeable in a variety of technical areas, not only because they're exposed to many technologies over time but also because they often face project pressures of getting disparate systems to work together.

The software development vocabulary is, of course, quite rich. In the big picture, it ranges from firmware and low-level device drivers to high-level applications, with various operating systems, command shells, editors, compilers, libraries, frameworks, protocols, and debuggers in between. Woven among those are more abstract elements such as algorithms, patterns, and programming language idioms.

Despite the fact that this breadth and rich variety has been the norm in software development for years, many developers continually try to avoid it by sticking with one editor or one operating system. Nowhere is this phenomenon more pronounced, however, than in the area of programming languages. Rather than choosing the best language for the task at hand, many developers (even those working on integration) try to bend the problems they face to fit their favorites. The results are often vastly inferior to what they might have been if the developers had chosen the right languages to begin with.

Increased Productivity

Knowing and using multiple programming languages for normal day-to-day development can yield significant benefits. No single language is a great fit for all problems. A programming language usually owes its existence to one simple fact: its designer felt it could address a set of problems — perhaps even just one problem — better than other available languages. This belief is apparently not uncommon: thousands of languages have come and gone and thousands more will follow. Numerous trade-offs are involved in programming language design and development, so there's room for many different approaches and variants.

Unsurprisingly, monolingual developers tend to choose general-purpose rather than specialized programming languages. General-purpose languages perform adequately for a wide variety of problems, but they generally yield predominantly middle-of-the-road solutions — neither great nor terrible. Of course, some monolingual developers possess extremely deep and thorough knowledge of their programming languages, and so know how to exploit them to the fullest. Yet, even such power programmers are constrained by the languages' practical limits.

It might be technically possible to solve a text-processing problem with a programming language designed primarily for number crunching, but nobody would argue that it's a good choice. There's little point in putting in the extraordinary effort needed to extract an unnatural solution from a language that's not designed for the problem at hand. This number crunching versus text processing example might be obvious, but it clearly indicates that language choice is an impor-

 1089-7801/08/$25.00 © 2008 IEEE

tant factor in developer productivity. Most scenarios that involve choosing the best language for a given problem are not as obvious; nevertheless, productivity increases that stem from choosing the right language can be significant and worthwhile.

Because general-purpose languages are often "good enough" for a wide variety of problems, monolingual developers tend to fall into ruts and never even consider other options. Consider XML processing, for example. For many languages, it can introduce a lot of accidental complexity — additional overhead associated with the solution rather than the problem — because of the inherent impedance mismatch between constructs available in XML and those provided by typical general-purpose languages such as Java and C++. Many developers simply put up with the mismatch and slog their way through, eventually reaching what is, at best, a mediocre solution. To help with productivity issues, they often resort to code generation, mapping XML constructs to statically typed programming language constructs to try to ease the impedance mismatch.

Unfortunately, that approach can be extremely brittle as a result of converting highly flexible XML constructs into rigid static data types that are difficult to version adequately. Any changes to the XML document then require new code generation to reflect those changes, even if the application doesn't use the specific modified XML entities. The newly generated code can, in turn, require changes to the application code that uses it, so that any application using the generated code must undergo full build, test, and redeployment cycles. Any minor productivity gains achieved through code generation are quickly lost in the noise when compared to ongoing maintenance costs.

Contrast this story of XML development — unfortunately, repeated quite often in enterprise-integration scenarios — with simply using a programming language that's better suited to the task. For example, the Python language `xml.etree` module makes XML handling almost trivial (even with versioning), and Perl has XML packages that are equally easy to use. Erlang's `xmerl` module is quite good as well. Better still, though, are languages that support *literal* XML, such as ECMAscript for XML (E4X) and Scala, which both let developers write XML directly within the language's syntax. Literal XML effectively eliminates the impedance mismatch between XML and the programming language, letting the developer write just a few lines of code versus what might require hundreds or thousands of lines in a combination of generated and manually written brittle Java or C++ code.

Easier Maintenance

Far from being limited to initial development, productivity gains from choosing the right language are even more pronounced in the code-maintenance phase, the span of which, for a successful long-lived system, far exceeds the time required to first develop it.

In *The Mythical Man-Month: Essays on Software Engineering*,[1] Fred Brooks cites several studies showing that the effort required to develop and maintain software rises exponentially with the number of instructions. He also explains that this phenomenon appears to be independent of the programming language in use. Given the exponent of 1.5 that he specifies, a program with three times as many lines of code as another program requires more than five times as much effort to develop and maintain. With five times the number of lines, the level of effort increases 11 times, and with 10 times as many lines, development and maintenance take a whopping 32 times the effort.

Extension and maintenance are areas in which the benefits of choosing the right language really shine. In part, this is because the right language lets developers provide initial solutions quicker, thus putting applications into users' hands that much sooner. Users can then provide quicker feedback and enhancement requests, which the developers can, in turn, service quicker — not least because using the right language means fewer lines of code to modify or augment. This process can become a cycle of positive reinforcement, in which fewer lines of code result in fewer defects and easier enhancement, which leads to happier users who provide free word-of-mouth advertising along with better feedback that helps improve the software even further.

Those who disagree with Brooks typically claim that modern integrated development environments (IDEs) and other tools invalidate the results he cites, but I'm unaware of any formal studies or publications to that effect. Judging from my own personal experience, IDEs can certainly enhance productivity, but only for particular languages such as Java and Smalltalk. This means they're often only partially useful, and sometimes not useful at all, to multilanguage developers. Although I've definitely seen developers display much higher productivity than others they worked with, in my experience it's never been only because those developers used IDEs while the others didn't.

Whether or not you use an IDE, an application's size has a tremendous impact on its development and maintenance costs. The larger the application, the more developers it takes to fully understand its architecture, design, and implementation, and the less likely it is that any single developer can visualize and memorize the whole system at once. As the number of these core developers needed to completely comprehend the system grows, the number of communication channels between them increases exponentially. The more of these

paths there are, the harder it is to ensure that code changes to the system are appropriate and correct.

Brevity, therefore, matters a great deal. If any of several languages could deliver acceptable performance, scalability, throughput, and other relevant characteristics, but one of them required an order of magnitude fewer lines of code, that's the one you'd want to choose.

Integration

One issue that comes up when considering multilanguage programming is how to get the different languages to work together. Fortunately, this isn't nearly as problematic as it might seem, especially within enterprise integration environments. Integration often implies distribution, which means that various parts of the system communicate through network messages. The network thus provides natural boundaries between components, allowing developers to use the most suitable programming language to write each component.

For non-networked multilanguage integration, virtual machine development has moved from targeting single languages to supporting multiple languages. For example, the Microsoft Common Language Runtime (CLR) supports a growing number of languages and enables languages of vastly different types — imperative, functional, and dynamic or "scripting" languages, for example — to interwork. Similarly, the Java virtual machine (JVM) has evolved from a Java-only platform to a base for a variety of languages, including JRuby, Scala, Groovy, JavaScript, E4X, Jython, and many others.

Given the JVM's evolutionary path as a multilanguage platform, I find it ironic that the Java community seems to have more than its share of fanatical monolingual developers. It costs essentially nothing for a Java developer to use another JVM-based language, other than the time and effort to learn the other language. Because all JVM-based languages are built on the same underlying bytecode, Java code can call into them, and they can call Java code and all existing Java libraries. The JVM thus provides a virtually pain-free way to mix and match the best languages for each part of an application.

Barriers

I've heard that some Java developers avoid using other languages because their management simply demands it. Such managers believe that they can more economically develop and maintain a wide variety of solutions by sticking to a single general-purpose popular language such as Java. They consider their developers to be commodities that they can easily interchange and replace because finding Java programmers is (at least for now) relatively easy. Thus, they believe Java-only development protects them from being stuck with code that only a few experts know how to read or maintain.

Managers who make such choices fail to consider all the costs involved in software development and maintenance. Allowing the use of better-suited languages — especially those that are JVM- or CLR-based — could easily lower a system's overall cost across its lifetime by reducing its size and thus the effort required to work on it. Smaller systems require fewer developers, which can mean significant short- and long-term cost savings.

Some developers claim it's just too hard to learn new languages, and that the time you spend wallowing in mediocrity as you perfect your newfound skills would be better spent working with the language you already know. For those who've struggled to learn a single general-purpose language such as Java or C++, the very prospect can be daunting because they expect all languages to be just as large and complicated. Fortunately, languages such as Lisp, Python, and Erlang are relatively simple in terms of core concepts, so beginners can be productive with them very quickly. Yet, the apparent simplicity of such languages belies a richness that can keep hardcore language enthusiasts busy for years discovering hidden treasures.

I would never claim that learning a new language is easy, but I can say from experience that the more languages you learn, the easier it becomes to learn yet another. Each new one you learn also helps improve your skills with those you already know: you tend to better understand the core concepts, which helps you see improved ways of using each language.

If you work on integration projects, one way to get started on a new language is to choose an "edge" system to develop in that language. Take some relatively isolated client code, for example, and re-implement it in a language that you think, based on research and some light experimentation, could make it smaller and easier to maintain. Focus not only on learning the new language's best practices but also on how best to integrate it with the rest of the system. If you succeed there, move across the network and try something on the server side. Whatever you do, don't be afraid of failure, because failing can be a very productive way to learn.

After all, do any of us really believe we've already learned the last programming language we'll ever need? ◻

Reference

1. F.P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, 2nd ed., Addison-Wesley, 1995.

Steve Vinoski is a member of the technical staff at Verivue in Westford, Massachusetts. He is a senior member of the IEEE and a member of the ACM. You can read his blog at http://steve.vinoski.net/blog/ and reach him at vinoski@ieee.org.