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

Java Makes Scripting Languages Irrelevant?

Diomidis Spinellis

Simplicity does not precede complexity, but follows it. — *Alan J. Perlis*

In computing, we often solve a complex problem by adding another level of indirection. For example, in Unix file systems, an index node, or *inode*, data structure lets the operating system allocate files concurrently and sparsely while still providing an efficient random-access capability. When we want to customize large, complex systems or express fluid, rapidly changing requirements, we frequently add a scripting layer on top of the corresponding system. Back in 1962 to 1963, Dan Murphy did this when he developed his TECO (*t*ext *e*ditor and *co*rrector) editor on the Digital Equipment Corp.'s PDP-1; its command language also doubled as an arcane (to put it politely) macro language.

Twenty years later, adding a scripting-language interface to existing applications (often written in C) was all the rage. Lotus 123 supported macro commands, Framework had the FRED (*fr*ame *ed*itor) language, and you could program AutoCAD and Emacs in a form of Lisp. On the Unix platform, administrators wrote sophisticated sendmail configuration files to bridge email networks—which were then disparate and mutually incompatible. John Ousterhout completed the picture by developing Tcl/Tk as a general-purpose scripting language for integration with any system that could benefit from such a capability. A few years later, Microsoft came up with Application Basic as its general-purpose scripting language for all its office productivity applications.

Those early developments acquainted programmers with the notion of customizing applications through scripting and opened the road for powerful, general-purpose scripting languages such as Perl, Python, and Ruby (see John Ousterhout's article, "Scripting: Higher-Level Programming for the 21st Century," in *Computer*'s March 1998 issue). Scripting languages glued to applications serve an important purpose: they greatly ease the application's configuration and customization and support end-user programming by offering a safe and friendly development environment. Gone are the intricacies of C's memory management, the convoluted string manipulation, and the complexity of the application's internal data structures. Instead, the scripting language typically offers automatic memory management, a powerful string data type, sophisticated data structures, a rich repertoire of operations, and an intuitive API for manipulating application data and state. Additionally, by interpreting the scripting language, the application can isolate itself from undesirable effects of the scripting code, such as crashes and data corruption.

My impression is that with the evolution of Java and Microsoft's .NET offerings (I'll use the term Java as a stand-in for both alternatives), the niche occupied by scripting languages is rapidly shrinking. We are approaching the end of an era.

0740-7459/05/$20.00 © 2005 IEEE

Rumors of the death of scripting languages ...

Java now offers most of the nice features that scripting languages provide to applications:

- automatic memory management through garbage collection,
- a standard string data type,
- collection interfaces implementing most useful data structures, and
- a very rich language library.

Additionally, in applications written in Java, what we might consider an API already comes for free as part of the object-oriented design. You only need to allow an application to dynamically load user-specified classes, expose its API by providing access to some of the application's objects, and limit the undesirable couplings through the security manager and exception handlers, and the need for a separate scripting language vanishes.

Many modern Java applications that support beans, plug-ins, and other extension mechanisms follow exactly this strategy—notably, Eclipse, Maven, Ant, Javadoc, ArgoUML, and Tomcat. Even on resource-constrained embedded devices—such as mobile phones, which are still programmed in a system programming language—configuration and customization is moving in a Java direction.

... are greatly exaggerated

Does the trend of customizing applications through a Java interface make scripting languages irrelevant? Yes and no. As an application configuration and extension mechanism, Java is probably the way to go. The cost of marshaling and unmarshaling data objects and types between the application's code written in Java and the conventions expected by a different scripting language is too high for the limited incremental benefits that the scripting language would offer. On the other hand, scripting languages still have an edge in a number of areas, offering us many distinct advantages.

A more flexible or imaginative syntax. Think of Perl's numerous quoting mechanisms and its regular expression extension syntax or Python's use of indentation for grouping statements. These make some program elements a lot easier to read. As an example, variable substitution within Perl's or the Unix shell's double-quoted strings is by far the most readable way to represent a program's output.

Less fuss about types. Most scripting languages are typeless and therefore easier to write programs in. For example, Perl makes writing a client or server for an XML-based Web service a breeze, whereas in Java we have to go through numerous contortions to implement the same functionality. Of course, the robustness and maintainability of code written in a typeless language is a different question, as many of us who maintain production code written in a scripting language later discover.

A more aggressive use of reflection. Consider Perl's `eval` construct and Python's object emulation features, which let programmers construct and execute code on the fly or dynamically change a class's fields. In many cases, these features simplify the construction of flexible and adaptable software systems.

Viability as a command language. Many scripting languages (such as those in operating system shells) can also double as a command language. Command-line interfaces often offer a considerably more expressive working medium than GUI environments (I'll expand on that in another column). Coupling a command-line interface with a scripting language means that you can easily promote commonly executed command sequences into automated scripts—a boon to us developers. This coupling also encourages an exploratory programming style, which many of us find very productive. I often code complex pipelines step by step, examining each step's output, before tacking another processing element at the pipeline's end.

A shorter build cycle. For many systems, a build cycle that provides time for an elaborate lunch is now sadly history. However, the tight feedback loop permitted by the lack of a compilation step in scripting languages allows for rapid prototyping and exploratory changes, which can often occur hand-in-hand with the end user. This is a feature that those using agile development methodologies can surely appreciate.

Does the trend of customizing applications through a Java interface make scripting languages irrelevant? Yes and no.

So, where do we stand now? The gap between system programming languages and scripting languages is slowly closing. For example, some scripting languages are capitalizing on Java's infrastructure by having their code compile into Java virtual machine bytecode. However, a lot of ground in the middle is still up for grabs. New system programming-language designs can offer more of the advantages now available only through scripting. And scripting languages are constantly benefiting from hardware performance advances that make their (real or perceived) efficiency drawbacks less relevant. The issue of the result's quality remains an open question on both fronts.

We developers, as avid tool users, can enjoy viewing the battle from on top and reap the benefits.

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 Reading: The Open Source Perspective* (Addison-Wesley, 2003). Contact him at dds@aueb.gr.