Editor in Chief: Hakan Erdogmus ■ National Research Council Canada
hakan.erdogmus@computer.org

So Many Languages, So Little Time

Hakan Erdogmus

What's up and coming in the programming language arena? A rudimentary analysis of 200+ sessions' titles and abstracts at OOPSLA '07 (22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications) provides a rough idea.

About one-third of the conference's total airtime addressed topics related to a distinguishable language paradigm. I grouped the language coverage into five focus areas addressing different language paradigms, then estimated each area's percentage of conference airtime. Here are the results:

- Objects and object-based approaches: 67%
- Functional languages and programming: 15%
- Dynamic languages: 10%
- Domain-specific languages: 6%
- Aspect orientation: 2%

I calculated airtime using a time-based weighting scheme, with a full-day event weighted at one. The five focus areas, however, weren't mutually exclusive or comprehensive. Many languages are multiparadigm and are becoming more so as they evolve.

These data represent only the conference's supply side. To capture demand, I also conducted an informal poll in the conference hallways (see figure 1). I asked 99 randomly selected delegates (about 8 percent of the attendees) seven multiple-choice questions. The questions probed respondents' perceptions regarding the top four focus areas.

Objects' ironic legacy

Frederick P. Brooks Jr. himself stated at the conference that if any technology deserves the "silver bullet" label, it would be objects. (See page 91 of this issue for a summary of the OOPSLA retrospective on "No Silver Bullet.") The poll respondents echoed his sentiment in large numbers, declaring that objects generally have been good for software development. However, when I asked whether most developers use objects properly, fewer than 7 percent said "Yes." Among those who believe they "get" objects, the sentiment that lay people don't is widespread. Mysteriously, a paradigm's mainstream status and positive impact don't correlate with informed usage. I wonder, if most people understood objects better and used them properly, would the technology have had even greater impact? I'm not sure.

Dynamic languages are getting hotter

Dynamic languages are all about runtime flexibility. Rather than putting a straightjacket on program behavior at compile time, dynamic languages support behavioral manipulation and adaptation at runtime and encourage fast, on-the-fly experimentation.

What constitutes a dynamic language? Most agree that Ruby, Perl, Python, Lua, PHP, Groovy, and JavaScript are truly dynamic. Despite its added late binding ability, not everyone considers VisualBasic dynamic because of its static typing roots. Of older-generation languages, Smalltalk, APL, Tcl, and Lisp are deemed dynamic. To many, a language paradigm is more a matter of usage style than an intrinsic language property. I suspect when answering the question "To what extent do you use a dynamic

Mission Statement: To be the best source of reliable, useful, peer-reviewed information for leading software practitioners—the developers and managers who want to keep up with rapid technology change.

 0740-7459/08/$25.00 © 2008 IEEE

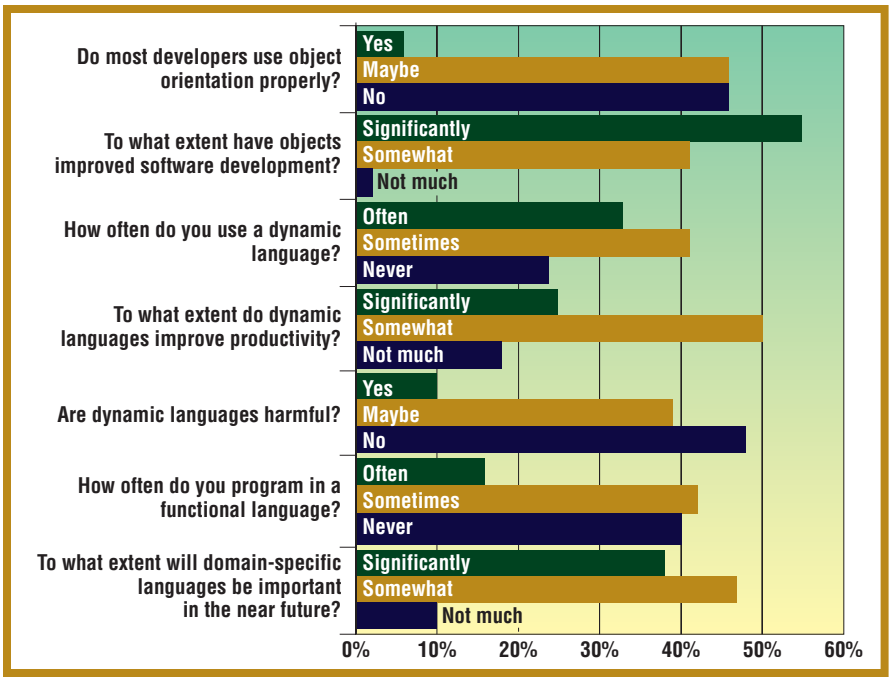

Figure 1. A sample group of OOPSLA '07 delegates' perceptions of language paradigms.

language?" the respondents had this looser interpretation in mind. More than 70 percent indicated they use a dynamic language at least some of the time, and even more associated dynamic languages with improved productivity.

The question "Are dynamic languages harmful?" invariably elicited a nervous chuckle or an "aha," whereas I expected appeals for clarification. The chuckles must have had something to do with dynamic languages' association with dynamic typing and, in turn, reduced safety. Only 10 percent thought dynamic languages are harmful; nearly half categorically disagreed. Of those who acknowledged the dangers lurking beneath, concerns about safety weren't a serious impediment: voluntarily or not, 75 percent go ahead and use a language they consider dynamic anyway. And good for them. Nearly half of the respondents who thought most developers at least sometimes misuse objects didn't have a problem trusting the same people with the extra power that a dynamic language bestows.

The rise of functional languages

Synergies further blur the already fuzzy philosophical lines that separate languages. Notably, besides objects, many dynamic languages emulate a functional-programming style more naturally than do static languages. A related thread linking the functional and dynamic worlds is terseness, an attribute that many developers cherish.

Graham Hutton defines functional programming broadly as a computational style that emphasizes the evaluation of expressions composed of functions and their arguments. In contrast, the imperative style relies on execution of commands that manipulate a global program state. As with dynamic languages, whether a programming language can be called functional is a matter of opinion and depends on the extent to which the language supports a functional programming style and its constructs preserve functions' mathematical properties. For purists, any language with constructs that produce side effects isn't functional. Others would classify, besides the archetypal examples Haskell and Miranda, impure languages such as Standard ML, Scheme, Erlang, Objective Caml, and F# as functional.

Functional programming has been a territory frequented mostly by theorists and researchers. Barring a few examples (notably Ericsson's Erlang in the telecommunications domain), industrial experience with it has been limited. Philip Wadler, in his 1998 editorial "Why No One Uses Functional Languages" (*ACM SIGPLAN Notices*, vol. 33, no. 8, pp. 23–27), provides a long list of reasons. The list includes poor interoperability with mainstream languages, poor

IEEE Software

support by integrated development environments, lack of extensive libraries, instability, installation difficulty, lack of debuggers and profilers, lack of training, disinterest in software engineering methods, and lack of a track record in credible projects.

In recent years, some barriers have fallen. Undoubtedly, interest from the likes of Microsoft and Google has put functional programming on the practitioner's map. Other barriers, however, appear intact. I haven't seen much talk of development practices and methods geared toward functional programming. I don't know whether anyone has thought about test-driving functional programs or building in-process testing frameworks that respect and leverage functional-programming principles. The mathematical concepts underlying functional programming—monads, closures, lambda expressions, currying, comprehensions, and such—still intimidate programmers.

Still, things are looking up for functional languages. Perhaps Microsoft's drive to support LINQ (a functional-programming-based integrated query model), extend C# and VisualBasic with functional-programming-friendly constructs, and promote F# will bring functional programming within reach of .NET developers. My poll rather optimistically indicated that only 40 percent of OOPSLA delegates never use a functional language. The statistic seemed a bit low, even for a venue like OOPSLA, which attracts many academics and enlightened practitioners. One explanation is a too-liberal interpretation of "functional language." Even though I left interpretation to the respondents, clarification requests along the lines "Is X a functional language?" were nevertheless accompanied by informed comments such as "X can treat functions as first-class objects" or "I can use a functional programming style in X." Taken together with such comments, the results pointed to an awareness level I hadn't expected.

The future of domain-specific languages

Another productivity-motivated idea is domain-specific languages. Martin Fowler defines a DSL as a programming language "targeted to a particular kind of problem," in contrast to a general-purpose language "that's aimed at [solving] any kind of software problem" (http://martinfowler.com/bliki/DomainSpecificLanguage.html). DSLs leverage the targeted problem domain's particular vocabulary, constraints, and concepts through specialized environments, constructs, syntactic sugar, application programming interfaces, or a combination thereof.

DSLs have been attracting renewed attention for good reason, riding on the rising visibility of functional and dynamic languages. A noteworthy case is Google's Sawzall, a DSL for massively parallel data analysis. (Sawzall is an implementation, and progression, of Google's MapReduce programming model, which in turn is based on functional-programming concepts.) A resounding majority of OOPSLA delegates expressed renewed attention when I asked them about the role DSLs will play in the near future. We have yet to see how such high expectations will pan out.

A major enabler of domain specificity is the much touted amenability of hybrid languages to creating embedded DSLs. Examples are Ruby (a grassroots language supporting object orientation and dynamism) and Scala (a research language supporting OO and functional programming). For an instance of a DSL embedded in Ruby, see the September/October 2007 issue's focus section on dynamically typed languages.

The language wars may be over

Developers are warming up to the freedoms offered by multiparadigm languages. Improved interoperability through shared runtime environments, clever hiding of underlying mathematical concepts, and better IDE support and integration could well nudge them over the hump to explore new territory without entirely abandoning their home grounds. Those who dunk their toes could be rewarded by a world of possibilities previously unimagined. As Bedarra Labs' Dave Thomas put it, certain niche areas will probably remain unexplored by masses because of high entry barriers—most significantly, lack of fundamental skills.

The multiparadigm programming trend is a generalization of Erik Meijer's and Peter Drayton's motto, "Static typing where possible, dynamic typing when needed." As Meijer and Drayton suggest in their similarly titled paper (OOPSLA '04 Workshop on Revival of Dynamic Languages), the wars may be coming to an end, if they're not entirely over. What are your thoughts?