Panel

Designing the Next Educational Programming Language

Andrew Black
Portland State University
black@cs.pdx.edu

Kim. B. Bruce
Pomona College, CA
kim@cs.pomona.edu

James Noble
Victoria University of Wellington
kjx@ecs.vuw.ac.nz

Abstract

Object-oriented programming is widely taught in introductory computer science courses, however no existing object-oriented programming language is "the obvious choice" for a teaching language. This makes it harder to transfer skills, techniques, and teaching materials between courses and between institutions, and leaves employers uncertain what they should expect new graduates to know. We believe that the object-oriented programming languages community should take this opportunity to work together to select, shape, or design the next educational programming language, and propose a set of principles that the language should follow. The purpose of this panel is to start a dialog with the educational community to refine these principles and to consider next steps.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General

General Terms Object-oriented Programming, Language Design Principles

Keywords design, teaching, introductory programming

1. Introduction

In the 1980s, computer science and software engineering programs worldwide benefited from a surprising unanimity in their choice of programming language: Pascal. This unanimity benefited students and teachers, who were able to transfer their skills and techniques between institutions, and to use textbooks from a wide variety of sources. It was also of benefit to employers, who were able to rely on graduates having a shared basis in programming, and researchers, who had a *lingua franca* for presenting programs and their designs, analysis, and execution.

Onward! 2010 Educators' Symposium October 2010, Reno–Tahoe, Nevada, USA

With the wide acceptance of object-orientation, the choice of introductory language became much less clear. Smalltalk, the obvious choice, was at first far too resource-intensive for an educational environment. Educators had to choose between "toy" languages (like Budd's *Little Smalltalk* [2]) and proprietary extensions like Object Pascal and Objective C. Just when Smalltalk became a feasible choice, Java and C++ appeared on the scene, further fragmenting the educational community. Java remains the most popular choice, but it is now a large and complex language with many features for supporting professional software engineering, and bearing the scars of 15 years' evolution. New languages such as C♯ and Scala have benefited from what we have learned from Java, but they too are large languages, also designed to support professional practice, and retain duplicative features for backwards compatibility with C++ or Java. Some institutions are adopting Python as a first language; however, others are wary of an introductory language that lacks declarations for data fields and a static type system. Scheme, Haskell, ML, GBeta, Smalltalk, and C++ — to name but a few — have been or are being used as introductory languages in particular contexts, but none has gained the widespread acceptance of Pascal or Java.

The programming languages community has successfully addressed this problem in the past. In the 1950s, there were a large number of attempts to produce an "algorithmic language": these attempts were unified and gave rise to ALGOL-60 [13, 14]. In the 1980s, functional programming languages were similarly diverse: these differences were resolved by designing Haskell [9]. In the 1990s there was at least one attempt to produce an object-oriented teaching language, Blue [10, 11], but it did not succeed, partly because many faculty wanted to use an "industrial-strength" language in their courses, and at that time Java seemed like a viable alternative. Given the last 20 years experience using C++, Java, and other "real" languages for teaching, perhaps it is time to reconsider and instead create a language designed for novices?

2. Principles

We have yet to begin sketching a language design. Instead we propose the following principles to guide the selection or design of the next educational programming language.

Paradigm The language should support object-oriented programming, because object-oriented languages are widely used in teaching, practice, and research. The language should use garbage collection, because machine-level issues such as storage management are a distraction for novices and should not intrude on teaching introductory programming [11**?**]. The language should have a purely functional subset, because the functional style is becoming increasingly important in teaching programming [6] and functions operating on immutable objects make it much simpler to support concurrency and distribution [5, 7].

Simplicity The language should be simple to learn and simple to use. The language should provide one "fairly clear way" to do most things. Like Modula-3, the language specification should follow C.A.R. Hoare's fifty-page rule [4]. The language should not build-in features that can be satisfactorily added through libraries [1].

Teaching Language The primary goal for the language is introductory teaching, covering at least the first year of study: program design, data structures, and algorithms. Ideally, the language should also be suitable for teaching intermediate topics also: software craft, personal development practices, and software design. If the language finds more advanced uses and employment outside the classroom, that will strengthen the argument to use it for teaching; however, advanced use is not a *primary* design goal.

Language Levels The language should support progressive teaching strategies, which use language subsets and extensions that can be matched to students' experience [6]. For example, one course could start with top-level functions, adding objects, types, mutable state, failure handling, and modules in separate language components. Teaching subsets can also allow error messages to be tailored to suit different levels of experience with the language.

Adaptable The language should support a range of curriculum approaches, including at least objects-first, imperative-first, functions-first, and breadth-first. Particularly because pointers are difficult to learn, the language must support a clear model of object references.

Best Practice The language should capture current best practice in programming and program design. Where possible, common bugs (i.e., those typically illustrated in introductory programming classes, or detected by findbugs) should be prevented by design.

Unsurprising The features of the language should be unsurprising. To quote C.A.R. Hoare again: *"the job of the language designer is consolidation"* [8]. As much as possible, programmers literate in two or more current industrial languages should be able to guess the meaning of code in the new language.

Evidence As far as possible, the design of the language should be based on evidence about features of existing languages — empirical results, formal studies, and teaching experience.

Ease of Implementation A graduate student should be able to construct a simple implementation of the language in six months. An experienced team should be able to construct an optimized, robust implementation in less than a year. Language implementations may be interpreted, compiled statically, or compiled dynamically.

Type System The language should be strongly typed: that is, there should be no unchecked run-time type errors. The language should support teaching with both static and dynamic type systems [3]. The execution of the language should not depend on a program's static types.

Formal Semantics The language should lend itself to reasoning about programs, and potentially to verification, especially as verification and checking technology makes its way into introductory IDEs. For these reasons, the language should have a well-defined formal semantics.

Portable The language and libraries should be independent of implementation technology and infrastructure. The next language should be able to be used on Unix/Linux, Windows, Mac OS X, and directly over the web. Most likely the language should run on top of common existing virtual machines (JVM, CLR, JavaScript).

Concurrency and Parallelism The language must support teaching concurrent and parallel programming, without mandating one particular approach. As a minimum, the language must support teaching message passing and shared memory concurrency, perhaps in different extensions.

Development Environment The language must be supported by novice-friendly development environments (such as BlueJ or Racket) and good debuggers.

Graphics and Multimedia The language should support programming with graphics and multimedia, and event-driven programming. Several teaching approaches rely on graphics and multimedia programming; they are also important application areas in their own right.

Software Engineering The language should support teaching good software engineering practices. This means that the language will need some support for software modularity, for handling failure, and for programming to interfaces. The language should provide explicit support for preconditions, postconditions, and invariants that would be automatically checked during (or before) program execution.

Performance Model The language should support a simple performance model for simple programs. The language must be able to support teaching students "how to predict, control, and/or explain the performance of their programs" [12].

Efficiency Efficiency is not a concern of this language design.

3. Call to Action

We call on the object-oriented programming language community to design a new language to meet this need.

We invite the community to bring together a relatively small group to investigate features to be included, and to solicit feedback on the language as the design progresses. We believe that, to promote and maintain a coherent vision, the core language design must remain vested in a small group. However, the language design process should be as open and transparent as possible, and the resulting language specification should be vested in the community.

4. Panel

Members of the panel will lay out the need for an educational programming language designed for teaching novices object-oriented programming, and trace the history of earlier attempts to design educational languages. The panel will focus on many of the principles that have been agreed to and why they are important for an educational language, as well as how the language design will proceed.

Supporters

In addition to the three authors of this document, the following people support this effort.

- Gilad Bracha, Ministry of Truth.
- John Boyland, University of Wisconsin-Milwaukee.
- Sophia Drossopoulou, Imperial College, London.
- Susan Eisenbach, Imperial College, London.
- Michael Kölling, The University of Kent.
- Doug Lea, SUNY Oswego.
- Jan Vitek, Purdue.

References

[1] A. P. Black, E. Jul, N. Hutchinson, and H. M. Levy. The development of the Emerald programming language. In *History of Programming Languages III*. ACM Press, 2007.

[2] T. Budd. *A Little Smalltalk*. Addison-Wesley, 1987.

[3] L. Cardelli. *Handbook of Computer Science and Engineering*, chapter Chapter 103: Type Systems. CRC Press, 1997.

[4] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 reference manual. Technical Report Research Report 53, DEC Systems Research Center (SRC), 1995.

[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[6] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How To Design Programs*. MIT Press, 2001.

[7] B. Goetz, T. Peierls, J. Block, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison Wesley Professional, 2006.

[8] C. Hoare. Hints on programming language design. Technical Report AIM-224, Stanford Artificial Intelligence Laboratory, 1973.

[9] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *History of Programming Languages III*, pages 12–1–12–55. ACM Press, 2007.

[10] M. Kölling and J. Rosenberg. Blue — a language for teaching object-oriented programming. In *ACM Conference on Computer Science Education (SIGCSE)*, 1996.

[11] M. Kölling, B. Koch, and J. Rosenberg. Requirements for a first year object-oriented teaching language. In *ACM Conference on Computer Science Education (SIGCSE)*, 1995.

[12] D. Lea, D. F. Bacon, and D. Grove. Languages and performance engineering: Method, instrumentation, and pedagogy. In *SIGPLAN Workshop on Programming Language Curriculum*, 2008.

[13] P. Naur. The European side of the development of ALGOL. In *History of Programming Languages I*, pages 92–139. ACM Press, 1981.

[14] A. J. Perlis. The American side of the development of ALGOL. In *History of Programming Languages I*, pages 75–91. ACM Press, 1981.