

Addressing the Disconnect Between the Good and the Popular

Michael Hind

IBM T.J. Watson Research Center
hindm@us.ibm.com

Abstract

For several decades universities have taught programming languages as a fundamental part of their undergraduate curriculum. These courses cover the core topics used in the design of good programming languages. However, widely used commercial languages quite often seem to go against the conventional wisdom of good language design that is taught in these courses. This disconnect between what is taught as good language design and what languages are used in industry has put the programming language course in a bind. Specifically, as computer science departments feel the increased pressure to add new emerging topics, many departments are choosing to remove the programming language course from the core curriculum.

In this position paper, we argue that the disconnect between good language design and industry practice is exactly why a programming language course should be a crucial ingredient in any undergraduate computer science education.

1. Introduction

The traditional undergraduate programming languages course covers topics that provide a strong foundation for understanding the design of good programming languages. These include the topics of syntax, semantics, types, control structures, data abstraction and many others. Instructors typically illustrate these design choices by comparing how the choices are made by different existing languages. This pedagogical technique has two advantages: it solidifies the underlying concept with concrete examples, and it exposes students to different programming languages.

Given this foundation of sound programming language design, one would expect the widely used commercial languages to embody these principles. Although there are examples where this is the case, more often it seems that the widely used programming languages are not the ones held up as examples of good design in the programming languages course. Thus, there appears to be a disconnect between what languages industry should use, according to what is taught in the programming languages course, and what are actually used.

Meanwhile as the computer industry evolves, new topics need to be covered in the undergraduate curriculum. These emerging topics put pressure on computer science departments to balance their

curricula; they need to remove material to enable room for the new material. Because of the above mentioned disconnect, department-wide curriculum committees do not see the priority in teaching the fundamental concepts covered in the programming languages course. Thus, this course is often the victim in the struggle on what material to cut from the curriculum.

However, it is our belief that this disconnect between the “good” languages surveyed in a programming languages course and the widely used languages provides an excellent opportunity to argue for the importance of a programming language course. We elaborate on this idea in the remainder of this paper.

2. The Good and the Popular

Anyone who has participated in a programming language course gets the impression that the field has matured and there are well-known principles for designing a programming language. However, when one looks at the popularity of languages, such as C, Visual Basic, or Perl, that seem to not be great examples of good general-purpose programming language design, one has to wonder how important is such good design, in practice. Just like the proponents of supposedly superior Betamax technology, one has to wonder if the “wrong” side is winning the battle.

This large disconnect, between what languages should be used, and what languages are used, poses an important question for programming language educators. Specifically, *why are the popular languages popular?*

We feel it is the obligation of the programming language course to explain this conundrum. One way would be to enumerate the other factors that are involved in the adoption of a programming language, and to help provide the tools to assess not only the strengths and weaknesses of future languages, but their likelihood for widespread success. If anyone can understand and explain this phenomena, it is the experts in the programming language area.

To illustrate, one possible explanation, let’s consider the various design principles that are often described in studying a programming language. Many textbooks refer to such attributes as

- *writability*: how easy is it to construct new programs,
- *readability*: how easy is it to understand programs,
- *maintainability*: how easy is it to modify existing programs to add new features or correct defects.

Often the point is stressed that maintainability is very important in industry because programs are maintained for a much longer time than they are initially written. However, scripting languages, like Perl, do not seem to follow this mantra, but nevertheless are widely used in industry. Instead they emphasize writability over maintainability. This is likely due to market forces, where programs needed to be written in a quick manner to capture a market op-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

portunity, and issues such as maintainability and performance are secondary.

We feel the programming languages course should tackle issues like this directly to help students understand why popular languages are popular and help predict if such languages are a fad or if they will endure. The alternative option of assuming that the “good” languages will eventually overtake the popular “bad” ones is dangerous and will lead to the further marginalization of the undergraduate programming languages course.

3. Stepping Back

The thesis of this position paper is a simple, non-radical one. It does not advocate a major redesign of the fundamental concepts of the programming language course. Instead it challenges such courses to explain the disconnect between many popular language and the “good” languages taught in a programming language course. This can be achieved in many ways. We leave it to the pedagogical experts to elaborate on the details.

Other Thoughts

Before writing this position paper, I polled various members of my company on the question of what should be taught in the undergraduate programming language course. I received replies from developers, recruiters, researchers, and executives. In most cases, their feedback was on broader questions than the programming language course. I will share a few of these ideas below. Obviously, this feedback is not an official position of IBM.

- *Students should understand the performance impact of using a particular language construct.* I cautioned that this can be very dangerous. For example, a decade ago, one may have avoided the use of virtual calls, but modern implementations render such calls nearly as efficient as direct calls.
- *Students should have a survey course of as many modern languages as possible.*
- *More rigorous high level programming material should be taught. This includes logic, set theory, and automata.*
- *Students should have experience working in small group on projects.*
- *Students should have an understanding that code is not standalone, it needs to be integrated into an environment and often needs to comply with outside requirements, such as governments.*
- *Students should think more about the problem being solved rather than the code to do it.*
- *Students should be taught languages in the proper context, for example, C and C++ should be used for systems programming.*