

A Project-Based Approach to Programming Language Evaluation*

James Howatt
Department of Computer Science
St. Cloud State University
St. Cloud, MN 56301-4498

`jhowatt@StCloudState.edu`

Abstract

Most programming language evaluation criteria are based solely on characteristics inherent in a language. Software developers could better use evaluations that help them determine how well-suited a language is for their particular task. This paper proposes such an evaluation method.

Introduction. Answers to “Which programming language is the best?” can ignite fierce arguments among zealots who see no reason for any language other than the one they use. Fortunately, most programmers do not hold such extreme opinions, and often would like to see a rational evaluation of different languages from which they can draw their own opinions. Many researchers have proposed methods for comparing and evaluating languages [2, 9, 10], but they tend to focus more on the languages than on the needs of language users. Although languages are intrinsically worthy of study, their real purpose is as tools in problem solving.

Users do need to know the strengths and deficiencies inherent in a language, and how well a language applies to an application domain. But, even within an application domain, requirements for two distinct projects may vary widely. One product may have to be highly reliable and portable, while another may have to be extremely efficient. Thus, knowing how well a language supports an application domain may not suffice; we also need to know how well a language supports the needs of particular projects within a domain. We need to be able to evaluate languages for their applicability to a specific project. This paper proposes a language evaluation method with just that focus.

In the following sections, we review major categories of programming language evaluation criteria, and propose an evaluation scheme that could help software developers determine the best language for their particular task. Finally, we reflect on the significance of our proposal.

Language Evaluation Criteria. To help us better sort and understand proposed measures of language goodness, we settled on four major categories. Three contain criteria that could be applied even if humans never read or wrote a line of code. The fourth category contains criteria for those characteristics that enhance or impede use of a language by human beings. Certainly, some criteria

*This paper appears in *ACM SIGPLAN Notices*, July, 1995 (30,7) pp. 37-40.

fall in more than one category—modularity enhances understanding (human factors) as well as reusability (software engineering)—but we preferred to place each criterion just once. Our four categories are:

1. **Language Design and Implementation Criteria:** Included here are those criteria that assess how well a language is designed and how easily compilers/interpreters can be written for it. Questions posed by Hoare[6], Wirth[16], and Wasserman[14] include: Is the language formally defined? Is the language unambiguous? Can a fast, compact compiler be written to generate efficient, compact code?

Although most of these questions evolved from early language and compiler development, before the existence of compiler-writing tools, they are still as applicable as languages grow more complex.

2. **Human Factors Criteria:** These criteria are used to assess the human interface or the user-friendliness of a language. They help answer the question “To what degree does the language allow a competent programmer to code algorithms, easily and correctly, so they can be understood, easily and correctly, by other competent programmers?” (We assume a competent programmer will write code with the aim of making it understandable to others.) A second question to be answered is “How easy is the language to learn?” Since programming is human-intensive, it is not surprising that most proposed evaluation criteria fall in this category. Although most programming language texts cite some criteria, Fischer and Grodzinsky[3], Friedman[4], Ghezzi[5], and Sebesta[12], provide reasonably complete descriptions.

3. **Software Engineering Criteria:** These assess those aspects of a language that enhance the “engineering” of good software. Such criteria will rate a language’s capacity to support portability, reliability, maintainability, reusability and all the rest of the typical engineering concerns. Criteria motivated by managerial concerns—availability of good quality compilers and of experienced programmers—are included, as are criteria motivated by the current popular design and programming paradigms. Today, a language that supports object-oriented methods would probably rank higher than one that does not.

Good sources for software engineering criteria include Marcotty and Ledgard[8], Sammet[11] in her classic history of programming languages, Shaw[13], and Watt[15].

4. **Application Domain Criteria:** These criteria assess how well a language supports programming for specific applications. Unfortunately, most discussions of language evaluation either propose very general criteria, or fail to cover this category at all. Two teams of researchers, however, did define some criteria based on the designers’ intended use of the language.

Shaw and her colleagues [13] evaluated the software engineering characteristics of several languages by rating each language’s “core”—the image the designers of the language had about how they thought the language should be used. AlGhamdi and Urban [1] called this approach the “philosophy of the design”—the intent of the designers in designing the language. Other good sources of more general criteria include Klerer[7] and Watt[15].

Project-Based Evaluation That relatively few, and non-specific, application domain criteria have been defined should be no surprise. Each application domain has unique requirements. And, even if we were to develop a set of domain-specific criteria, not all of the criteria would apply to the same degree for each problem within the domain. Not every task requires each “good” attribute of a language. Therefore, perhaps instead of trying to rate the intrinsic “goodness” of a language, or trying to rate its applicability to a domain, we should rate a language as to how well it helps us solve the problem at hand. So, in addition to asking if a particular language possesses a particular attribute, we should ask how much that attribute applies to our particular problem. If a poorly-rated language’s deficiencies that do not apply to our project, then the language may not be a poor choice.

Therefore, it seems reasonable that criteria to evaluate or compare languages should be defined or selected by software developers during the specification or architectural design phases of a project. Project-based language evaluation methods would not only allow application experts to select their criteria, but also to specify the importance of each criterion to the application.

To assist developers, we could construct a checklist containing all proposed evaluation criteria. But instead of making it just a yes/no checklist, we would put it in a format like the following:

1. Criterion: a description of the quality to be measured
2. Degree to which the language satisfies that criterion
3. Degree to which the criterion is important to our project

Language developers and software engineers would assemble the list of criteria, language experts would provide the ratings for item 2, and the software project team would answer item 3.

Using such a checklist would help developers not only identify criteria to consider, but also to evaluate the criteria based on the requirements of their project, and thus, select a best language for their needs.

Some may argue that many language attributes are important to all development efforts, and the determination of their relevance should not be left to individual development teams. If this is true, then perhaps such criteria can be codified as generally accepted, or even mandated, software engineering standards. But use of the proposed checklist would move language evaluation from its current inherent-characteristics focus to one more directly applicable to software development.

Relevance. Even if such a language evaluation checklist were developed, it may not receive wide use. When selecting a programming language, most software development organizations choose the

So, perhaps developing a project-specific method of evaluating languages is of only academic interest—something to teach our students before they are faced with real-world constraints in software development.

Our approach then might be not to find a best language for a given task, but to use the one we would have used anyway, determine its strengths and weaknesses, and propose changes to enhance the strengths and to correct the deficiencies. Or, in short, to keep doing business as usual.

References

- [1] Jarallah AlGhandi and Joseph Urban. Comparing and assessing programming languages: Basis for a qualitative methodology. In *Proc. 1993 Software Applications Conference*, pages 222–229. ACM, Inc., 1993.
- [2] Alan Feuer and Narain Gehani, editors. *Comparing and Assessing Programming Languages*. Prentice-Hall, 1984.
- [3] Alice E. Fischer and Frances E. Grodzinsky. *The Anatomy of Programming Languages*. Prentice-Hall, 1993.
- [4] Linda K. Friedman. *Comparative Programming Languages: Generalizing the Programming Function*. Prentice-Hall, 1991.
- [5] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley and Sons, 2nd edition, 1987.
- [6] C. A. R. Hoare. Hints on programming language design. In *Proc. SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, 1973.
- [7] Melvin Klerer. *Design of Very High-level Computer Languages: A User-Oriented Approach*. McGraw-Hill, 2nd edition, 1991.
- [8] Michael Marcotty and Henry F. Ledgard. *Programming Language Landscape*. Science Research Associates, 2nd edition, 1986.
- [9] Michael L. Nelson. Considerations in choosing a concurrent/distributed object-oriented programming language. *ACM SIGPLAN Notices*, 29(12):66–71, 1994.
- [10] Research & Education Associates. *Handbook and Guide for Comparing and Selecting Computer Languages*, 1985.
- [11] Jean E. Sammet. *Programming Languages: History and Fundamentals*. Prentice-Hall, 1969.
- [12] Robert W. Sebesta. *Concepts of Programming Languages*. Benjamin/Cummings Publishing, 2nd edition, 1993.
- [13] Mary Shaw, Guy T. Almes, Joseph M. Newcomer, Brian K. Reid, and William A. Wulf. A comparison of programming languages for software engineering. *Software–Practice and Experience*, 11:1–52, 1981.

- [14] Anthony I. Wasserman, editor. *Tutorial: Programming Language Design*. IEEE Computer Society Press, 1980.
- [15] David A. Watt. *Programming Language Concepts and Paradigms*. Prentice-Hall International, 1990.
- [16] Nicklaus Wirth. On the design of programming languages. In *Proc. IFIP Congress 74*, pages 386–393. North-Holland, 1974.