Extended abstract

Interdisciplinary Programming Language Design

Michael Coblenz[a], Jonathan Aldrich[b], Brad A. Myers[c], and Joshua Sunshine[b]

a Computer Science Department, Carnegie Mellon University

b Institute for Software Research, Carnegie Mellon University

c Human-Computer Interaction Institute, Carnegie Mellon University

Abstract Researchers in the programming languages community are accustomed to reasoning about formal properties and performance characteristics. However, recent work has explored how user data can be incorporated into the language design and evaluation process to aid in the design of languages that make programmers more effective. In this paper, we describe an interdisciplinary collection of methods that have been used to incorporate user data into the language design and evaluation process, showing examples of how these have been used to improve various programming languages.

Keywords programming language design, user-centered design, programming language evaluation

The Art, Science, and Engineering of Programming

Submitted April 19, 2018

Published February 18, 2019

DOI 10.22152/programming-journal.org/2019/3/16

In *The Art, Science, and Engineering of Programming*, vol. 3, Essays, 2019, article 16; 3 pages.

This is an extended abstract of a paper that appeared in Onward! Essays 2018 [1].

Language development typically follows an iterative, two-phase process. First, the designer identifies requirements and creates artifacts, such as syntax and semantics specifications, compiler implementations, etc. Then, the designer must evaluate to what extent the language design meets the requirements. A typical process iterates between these two phases as the evaluation motivates additional design and implementation work. In this paper, we argue for the use of user-centered methods at all stages of development, incorporating both formative and summative methods to maximize the benefit for users while helping the designer by informing the design with insight from users.

While designing a language, the designer must obtain evidence regarding three kinds of properties. *Formal properties* capture mathematical facts about the language; *observational properties* concern real-world execution, such as performance characteristics; *effects on programmers* describe how language design decisions affect the humans for whom the language was created. Designers should obtain evidence through a collection of methods, summarized in the paper. For example, *corpus studies* can be useful to assess to what extent a potential problem occurs in the real world; qualitative user studies can be used to understand how some users behave and why.

The methods that are useful depend both on the phase of development and the kinds of properties that the designer is interested in reasoning about. Designers also bring their own perspectives and priorities; for example, an educator might focus on the pedagogical benefits of a language, whereas a logician might be primarily concerned with the language's formal properties. However, we argue that designers of languages that are most effective for their users combine perspectives and methods corresponding to several different properties and perspectives. Using multiple methods enables *triangulation*, which mitigates limitations of individual methods by using several complementary ones.

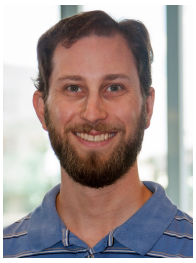
In the paper, we describe several language designs that have benefited from interdisciplinary language design methods and explain how they have done so: typestate in Plaid; transitive class immutability in Glacier; and AppleScript, which also incorporated user studies in its design methodology. We also describe situations in which designers missed opportunities to fruitfully incorporate user data, including in C and ALGOL.

References

[1] Michael Coblenz, Jonathan Aldrich, Joshua Sunshine, and Brad Myers. "Interdisciplinary Programming Language Design". In: *Onward! 2018 Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Nov. 2018). DOI: 10.1145/3276954.3276965.

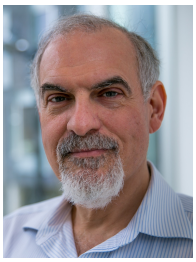
About the authors

Michael Coblenz is a fifth-year PhD student at Carnegie Mellon University, studying how to design programming languages that make software engineers more effective. Contact him at mcoblenz@cs.cmu.edu.



Jonathan Aldrich Jonathan Aldrich is a Professor in the School of Computer Science at Carnegie Mellon University. He develops new programming language designs that improve software engineering at scale. Contact him at jonathan.aldrich@cs.cmu.edu.

Brad A. Myers is a Professor at the Human-Computer Interaction Institute, Carnegie Mellon University. Contact him at bam@cs.cmu.edu.



Joshua Sunshine is a Systems Scientist at the Institute for Software Research, Carnegie Mellon University. Contact him at sunshine@cs.cmu.edu.