Usability of Programming Languages

Citation for published version:

Digital Object Identifier (DOI):
10.1145/2851581.2886434

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Usability of Programming Languages
Special Interest Group (SIG) meeting at CHI’2016

Abstract
Programming languages form the interface between programmers (the users) and the computation that they desire the computer to execute. Although studies exist for some aspects of programming language design (such as conditionals), other aspects have received little or no human factors evaluations. Designers thus have little they can rely on if they want to make new languages highly usable, and users cannot easily chose a language based on usability criteria. This SIG will bring together researchers and practitioners interested in increasing the depth and breadth of studies on the usability of programming languages, and ultimately in improving the usability of future languages.

Author Keywords
Programming language usability; API usability; end-user software engineering (EUSE); empirical studies of programmers; psychology of programming.

ACM Classification Keywords

Introduction
The empirical studies of programmers (ESP), which was also called the psychology of programming, dates back to before the CHI conference was formed (e.g., [20]), and yet programming is still a difficult human task. A human-centered definition says that “Programming is the process of transforming a mental plan into one that

Brad A. Myers
Human-Comp. Interaction Inst.
Carnegie Mellon University
bam@cs.cmu.edu

Andreas Stefik
Computer Science
University of Nevada, Las Vegas
stefika@gmail.com

Stefan Hanenberg
Dept. of CS and Business IS
Univ. of Duisburg-Essen, Germany
stefan.hanenberg@uni-due.de

Antti-Juhani Kaijanaho
Department of Math. Inf. Tech.
University of Jyvaskyla, Finland
antti-juhani.kaijanaho@jyu.fi

Margaret Burnett
School of EECS
Oregon State University
burnett@eecs.oregonstate.edu

Franklyn Turbak
Computer Science Department
Wellesley College
fturbak@wellesley.edu

Philip Wadler
School of Informatics
University of Edinburgh, UK
wadler@inf.ed.ac.uk

Margaret Burnett
School of EECS
Oregon State University
burnett@eecs.oregonstate.edu

Franklyn Turbak
Computer Science Department
Wellesley College
fturbak@wellesley.edu

Philip Wadler
School of Informatics
University of Edinburgh, UK
wadler@inf.ed.ac.uk

Permission to make digital or hard copies of part or all 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. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author(s).
CHI'16 Extended Abstracts, May 07-12, 2016, San Jose, CA, USA
ACM 978-1-4503-4082-3/16/05.
http://dx.doi.org/10.1145/2851581.2886434
is compatible with the computer” [11]. The programming language is the way that this transformation is expressed, and the smaller the transformation, the easier the programming task is likely to be [8].

However, few human factors studies provide guidance to language designers or users. In fact, a recent survey found only 22 randomized controlled trials (RCTs) of features of textual languages between the early 1950s through 2012 [12]. Even modern changes (e.g., Java with JDK 8 and 9, C++ 11 or 14, ECMAScript 6) have not been vetted from a human factors point of view.

A group of researchers working at the intersection of human-computer interaction (HCI), software engineering (SE), and programming language design (PL) are trying to provide appropriate methods for evaluating languages, as well as valid, empirically grounded evidence to guide design decisions. We hope that this can alleviate “programming language wars” [16] based purely on unsubstantiated claims. This special interest group (SIG) meeting will bring together these researchers, along with practitioners who have insights into usability issues for particular domains and situations, and programmers who want to evaluate languages they might use.

Examples of Methods
Researchers and practitioners have adapted a variety of conventional HCI methods for the study of programming language usability, and also have created entirely new methods. Examples of conventional methods include using randomized controlled trials (RCTs) [12] and longitudinal tracking of programmer behavior [1] to provide insight into language usability. Examples of new methods include the “natural programming” elicitation method [13], which tries to understand how people think about various concepts by letting them generate their own expressions for them (see Figure 1), and the “cognitive dimensions framework” which provides vocabulary to help language designers consider human-oriented programming language attributes at design time [8], and has been used to evaluate both textual and visual programming languages.

Examples of Results
There were many early studies about features of programming languages which made them difficult to learn for novice programmers (see a summary [14]). For example, a series of controlled experiments examined the usability for novices of certain variants of conditional statements [12]. Interpreting their results is not straightforward due to their heterogeneity; depending on the tasks given to study participants and on the chosen outcome measures, each of the variants was able to come on top. As discussed in detail in Sec. 10.3.1 of [12], considering only logic errors in programming as the outcome of interest, there is weak evidence in favor of conditional statements with a mandatory END token and repeating of the condition, a syntax not in current use. Similarly, studies of inheritance in object-oriented programming have shown both positive [3] and negative [5] effects on maintenance. Results from other studies showed that non-programmers naturally used condition-action expressions for events (such as “when Pacman hits a wall, he stops”) [13] (see also Figure 1). Figure 2 highlights user-study-based results in Alice [4].
There has been a long debate among programming language designers as to whether static or dynamic typing is more beneficial (for example, [2] argues strongly for statically typed languages while [18] argues the opposite), but only recently have scholars begun gathering evidence in a series of replicable experiments. Results under a variety of conditions (e.g., with/without a development environment, with/without documentation) show that developers are more productive with static typing (see, e.g., [7]).

With regard to notation, programming languages vary significantly in regard to the word/symbols chosen by the designers, the structure of the code, and the meaning of the notations. Recent studies have investigated whether these differences in notation matter. With novices, for example, results show that even minor changes to what word is selected (e.g., the word “repeat” instead of “for” in a loop, the use of “=” instead of “==” in an if statement) significantly impacts novices [17].

With regard to concurrency control, several controlled experiments indicated that software transactional memory was superior with respect to programmer time compared to locks [13, Sec. 10.3.2]. Another set of studies addressed the concept of aspect-oriented programming and generally found benefit only in the more complex cases for specific tasks [9]. Anonymous functions have also been studied (see Figure 3).

**Topics and Goals for the SIG**

In this special interest group (SIG) meeting, we focus on usability aspects of the programming language itself (rather than API usability, which was covered by a previous CHI SIG [6]). We are interested in appropriate techniques for measuring the usability of programming languages that focus on various aspects of usability, including the learnability, effectiveness, productivity, and error proneness of the language. We are also interested in techniques that go beyond lab studies, for example to measure the longitudinal impact of programming language design on professionals or learners. For all studies, we are interested in the details that make the results sound, valid and convincing.

For the language design itself, a first consideration is the overall presentation of the language – the usability of textual, visual, or hybrid languages. Also, the usability of kinds of languages – imperative, object-oriented, functional, constraint-based, etc. At a more detailed level, we are interested in studies of the usability of specific features, such as the syntax, keyword choice, special characters, the choice of static vs. dynamic typing, and advanced features such as concurrency and exception handling.

For all of these measures, we are interested in how they differ for different groups, such as learners vs. end-user programmers vs. professionals, gender differences, and the impacts on people with disabilities. Another dimension is the usability differences based on scale and size of the programs (from small to large to ultra-large), or target domain (e.g., scientific, concurrent, high-assurance, web programs, etc.).

One goal for this SIG is to provide a forum where HCI researchers who study programming languages can discuss appropriate methodologies and results. We hope this will increase the standard of evidence for studies of programming language design and codify “best practices” for usability evaluations of languages.
using namespace std;
float getSum(marketBasket mb) {
    float retVal = 0;
    function<void (item)> func =
        [&] (item theItem) {
            retVal += theItem.price;
        };
    mb.iterateOverItems(func);
    return retVal;
}

Figure 3: One recent study [19] found that anonymous functions in C++ 11 compared to iterators showed no benefits for developers from students through professionals. The study also found the C++ syntax for this new feature caused negative productivity impacts, alternative syntax or visual representations may harbor benefits, such as the graphical presentation in Snap! [10].

Acknowledgements
This work was partially funded by NSF grants IIS-1314356, CNS-1423054, CNS-1209457, IIS-1314384, CNS-1440878 and DUE-1226216, and by EPSRC EP/K034413/1. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of any of the funders.

and language features. A second goal is make this research more widely available by cataloging which features and demographics have been investigated, and which methods are most effective. We expect to also highlight where future research is needed. We will collect a bibliography of articles and blogs, along with venues where such articles may appear (such as PPIG, ICPC, VL/HCC, PLATEAU, OOPSLA, ICSE, CHASE, etc.) at the website: programminglanguageusability.org. Finally, we will discuss possibilities for a future forum on this topic, such as a possible future CHI Workshop.

References
12. Kaijanaho, A.-J., Evidence-Based Programming Language Design: A Philosophical and Methodological Exploration. PhD Diss., Information Technology Faculty, University of Jyväskylä, 2015, Jyväskylä, Finland. 222.