Usability of Programming Languages

Special Interest Group (SIG) meeting at CHI 2016

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

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

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

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

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

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

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

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

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
H.1.2 User/Machine Systems: Software psychology.
D.3.3 Language Constructs and Features.

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

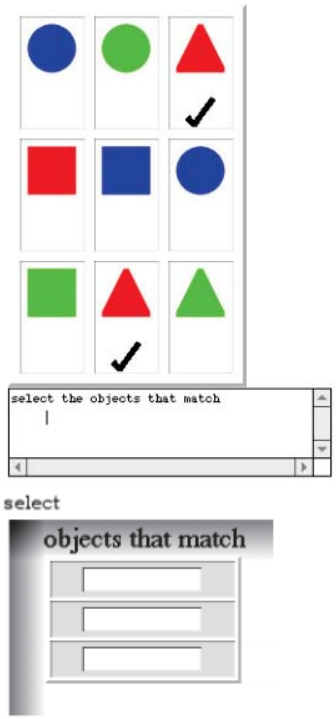

Figure 1: An early study that showed that non-programmers have more difficulty understanding and constructing queries using textual "and", "or" and "not" compared to using a tabular representation [15].

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].

```
obj.move(forward, 1)
obj.move(forward, 1,
   duration=3)
obj.move(forward, 1,
   speed=4)
obj.move(forward, speed=2)
# change of coordinate
   system
obj.move(forward, 1,
   AsSeenBy=camera)
# different interpolation
   function
obj.move(forward, 1,
   style=abruptly)
```

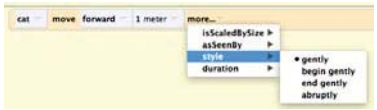

Figure 2: User studies for early versions of Alice that used Python as a scripting language found that Python's optional keyword parameters with defaults support a *controlled exposure to power* principle that facilitates the incremental learning of advanced Alice features by novices [4]. The drag-and-drop visual tile-based interface of more modern versions of Alice supports this principle through drop-down menus that include a "more" option for additional parameters.

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-1240957, 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

1. Altadmri, A., *et al.*, "37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data," in *SIGCSE'*2015. pp. 522-527.
2. Cardelli, L., "Type Systems," in *CRC Handbook of Computer Science and Eng., 2nd Ed.*, 1997, CRC Press.
3. Cartwright, M., "An Empirical View of Inheritance." *Inform Soft Technol* 1998. **40**(4): pp. 795-799.
4. Conway, M., *et al.* "Alice: Lessons Learned from Building a 3d System for Novices," in *CHI'2000*. pp. 486-493.
5. Daly, J., *et al.*, "Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software." *Empirical Soft. Eng.*, 1996. **1**(2): pp. 109-132.
6. Daughtry, J.M., *et al.*, "Api Usability: Chi'2009 Special Interest Group Meeting," *CHI'2009*, Boston, MA. pp. 2771-2774. See www.apiusability.org.
7. Endrikat, S., *et al.*, "How Do Api Documentation and Static Typing Affect Api Usability?," *ICSE 2014* 632-642.
8. Green, T.R.G., "Cognitive Dimensions of Notations," in *People and Computers* 1989, Cambridge Univ. Press.
9. Hanenberg, S. and Endrikat, S., "Aspect-Orientation Is a Rewarding Investment into Future Code Changes - as Long as the Aspects Hardly Change." *Information & Software Technology*, 2013. **55**(4): pp. 722-740.
10. Harvey, B. and Mönig, J., "Lambda in Blocks Languages: Lessons Learned," in *IEEE Blocks and Beyond Workshop*, 2015. pp. 35-38.
11. Hoc, J.-M. and Nguyen-Xuan, A., "Language Semantics, Mental Models and Analogy," in *Psychology of Programming*, 1990, Academic Press. pp. 139-156.
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.
13. Myers, B.A., Pane, J.F., and Ko, A., "Natural Programming Languages and Environments." *CACM*, 2004. **47**(9): pp. 47-52.
14. Pane, J.F. and Myers, B.A., *Usability Issues in the Design of Novice Programming Systems.* Technical Report, CMU-CS-96-132, August, 1996. Pittsburgh, PA.
15. Pane, J.F. and Myers, B.A., "Tabular and Textual Methods for Selecting Objects from a Group," in *IEEE VL 2000*. Seattle, WA. pp. 157-164.
16. Stefik, A., *et al.*, "The Programming Language Wars: Questions and Responsibilities for the Programming Language Community," *Onward! 2014.* pp. 283-299.
17. Stefik, A. and Siebert, S., "An Empirical Investigation into Programming Language Syntax." *Trans. Comput. Educ.*, 2013. **13**(4): Article 19.
18. Tratt, L. and Wuyts, R., "Dynamically Typed Languages." *IEEE Software*, 2007. **24**(5): pp. 28-30.
19. Uesbeck, P.M., *et al.*, "An Empirical Study on the Impact of C++ Lambdas and Programmer Experience," *ICSE 2016.* To Appear.
20. Weinberg, G.M., *The Psychology of Computer Programming.* 1971, New York: von Nostrand Reinhold.