Report from Dagstuhl Seminar 18061

Evidence About Programmers for Programming Language Design

Edited by

Andreas Stefik[1], Bonita Sharif[2], Brad A. Myers[3], and Stefan Hanenberg[4]

1 **Univ. of Nevada – Las Vegas, US,** stefika@gmail.com
2 **Youngstown State University, US,** shbonita@gmail.com
3 **Carnegie Mellon University – Pittsburgh, US,** bam@cs.cmu.edu
4 **Univ. of Duisberg-Essen, DE,** stefan.hanenberg@uni-due.de

Abstract

The report documents the program and outcomes of Dagstuhl Seminar 18061 "Evidence About Programmers for Programming Language Design". The seminar brought together a diverse group of researchers from the fields of computer science education, programming languages, software engineering, human-computer interaction, and data science. At the seminar, participants discussed methods for designing and evaluating programming languages that take the needs of programmers directly into account. The seminar included foundational talks to introduce the breadth of perspectives that were represented among the participants; then, groups formed to develop research agendas for several subtopics, including novice programmers, cognitive load, language features, and love of programming languages. The seminar concluded with a discussion of the current SIGPLAN artifact evaluation mechanism and the need for evidence standards in empirical studies of programming languages.

Seminar February 4–9, 2018 – https://www.dagstuhl.de/18061
2012 ACM Subject Classification Software and its engineering → Software notations and tools, Human-centered computing → HCI design and evaluation methods, Human-centered computing → Accessibility, Social and professional topics → Computing education
Keywords and phrases programming language design, computer science education, empirical software engineering, eye tracking, evidence standards
Digital Object Identifier 10.4230/DagRep.8.2.1
Edited in cooperation with Michael Coblenz

1 Executive summary

Michael Coblenz (Carnegie Mellon University – Pittsburgh, US)

Programming languages underlie and have significant impact on software development, especially in terms of the ability of programmers to achieve their goals. Although designers of programming languages can already reason about the *formal* properties of their languages, few tools are available to assess the impact of design decisions on programmers and software engineers.

At Dagstuhl Seminar 18061, a diverse set of participants gathered to review the existing body of evidence about programmers that has implications on programming language design.

Participants also reviewed existing research methods, such as eye tracking, that may help better understand the impact of language design decisions on programmers. Participants brainstormed a long list of possible research questions for investigation (§4), and then divided into working groups (§5) to focus on several areas of research interest, including novices, context switching and cognitive load, language features, emotional attachment to languages, and representativeness of subjects in studies. In each area, participants proposed research methods and questions that they felt would be valuable to address in the future. Then, the group discussed and prioritized these research questions.

The seminar included a discussion of the need for an evidence standard in empirical studies of programming languages, focusing on content of the evidence standard, adoption mechanisms, and criteria for what it might include in our field. Finally, the seminar concluded with a discussion of future directions for research, including a list of research questions that the participants were planning on collaborating on in the near future.

2 Contents

3 Overview of Talks

3.1 How do PL Researchers Think? and Evidence about Software Engineers for PL Design

Jonathan Aldrich (Carnegie Mellon University – Pittsburgh, US)

Joint work of Michael Coblenz, Joshua S. Sunshine, Brad A. Myers

Evidence about programmers is of increasing interest in programming language design, and my colleagues and I have argued for incorporating this evidence in an interdisciplinary approach to programming language design [1]. However, PL design is a domain quite different from other domains in which human behavior has been studied. This talk is an introduction, aimed at HCI experts and others studying human behavior, to the field of programming language design. I'll give an example of programming language design, discuss the goals and potential impact of language design, and talk about how PL researchers currently think about language design. I'll talk about what makes a programming language viable for large-scale development, and about how ideas from software engineering (which are ultimately derived from empirical research) could influence improve future language designs. Finally, I'll suggest some strategies that empirically-focused researchers can use to maximize their impact on PL design.

References

1 Michael Coblenz, Jonathan Aldrich, Joshua Sunshine, and Brad Myers. *Interdisciplinary Programming Language Design*. Working draft, available at http://www.cs.cmu.edu/~aldrich/papers/interdisciplinary-pl-design.pdf, February 2018.

3.2 How does the Programming Language Community Design a Language?

Craig Anslow (Victoria University – Wellington, NZ)

Designing a programming language is hard. The design process is complicated and different for each language. In this talk, the results from a study are presented from interviewing expert programming language designers (n=25) in the SIGPLAN community on how they designed their languages. The study also asked if they have conducted user experiments on their language and how they value these types of experiments. Results show that designing a language is very hard and a complex process; very few have adopted human-centered design methods or conducted user experiments. However, most language designers felt that conducting user experiments is of great value but very hard to perform. Encouraging programming language designers to adopt human-centered design methods and conduct user experiments will help to improve the usability and effectiveness of their languages.

3.3 Thinking about Programmers with Disabilities

Ameer Armaly (University of Notre Dame, US)

Programmers with disabilities face a variety of problems depending on the exact nature of their disability. Consequently, programmers with disabilities cannot be addressed as a monolithic group; researchers must instead examine the problems and impacts of individual disabilities. In this talk, I introduce the different ways disability has been thought of in society. I then present two research projects relating to programmers who are blind as examples of the different ways researchers can effectively examine and address the experience of programmers with disabilities. I also demonstrate in detail the experience of programming while blind and discuss ideas for future exploration.

3.4 Enhancing Compiler Error Messages: What's been done, what needs doing

Brett A. Becker (University College Dublin, IE)

Joint work of Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, Catherine Mooney
Main reference Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, Catherine Mooney: "Effective compiler error message enhancement for novice programming students", Computer Science Education, Vol. 26(2-3), pp. 148–175, 2016.
URL http://dx.doi.org/10.1080/08993408.2016.1225464

Programming is an essential skill that all computing students are expected to master. However, programming can be difficult to learn. Successfully interpreting compiler error messages is crucial for correcting errors and progressing toward success in learning programming. Yet these messages are often difficult to understand and pose a barrier to progress for many novices, with struggling students often exhibiting high frequencies of errors, particularly repeated errors [2]. The area of compiler error enhancement has seen increased activity in the last several years, particularly in presenting empirical evidence of effects. This talk focuses on these recent results. After discussing the importance of compiler error messages and the problems they present, empirical motivating evidence demonstrating a need for enhanced compiler error messages is reviewed. Then, recent empirical evidence on enhanced compiler error messages is discussed, including a debate on the effects and effectiveness of enhancing compiler error messages. In light of these recent results, the current state of play is then presented, organized in three areas: metrics and signals; compilers and languages; and frameworks, guidelines and principles. Finally, directions for research going forward are discussed.

References

1 Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill and Chris Parnin. *Do developers read compiler error messages?* Proceedings of the 39th International Conference on Software Engineering (ICSE 2017), Buenos Aires, Argentina, May 2017, IEEE Press. DOI: 10.1109/ICSE.2017.59
2 Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin and Catherine Mooney. *Effective compiler error message enhancement for novice*

programming students. Computer Science Education 26 (2-3), (2016) 148-175. DOI: 10.1080/08993408.2016.1225464

3 Brett A. Becker, Kyle Goslin, and Graham Glanville. *The effects of enhanced compiler error messages on a syntax error debugging test*. Proceedings of the 48th ACM Technical Symposium on Computer Science Education (SIGCSE 2018), Baltimore, Maryland, USA, February 2018. ACM. DOI: 10.1145/3159450.3159461

4 Brett A. Becker, Cormac Murray, Tianyi Tao, Changheng Song, Robert McCartney and Kate Sanders. *Fix the first, ignore the rest: Dealing with multiple compiler error messages*. Proceedings of the 48th ACM Technical Symposium on Computer Science Education (SIGCSE 2018), Baltimore, Maryland, USA, February 2018. ACM. DOI: 10.1145/3159450.3159453

5 Brett A. Becker. *An effective approach to enhancing compiler error messages*. Proceedings of the 47th ACM Technical Symposium on Computer Science Education (SIGCSE 2016), Memphis, Tennessee, USA, March 2016. ACM. DOI: 10.1145/2839509.2844584

6 Brett A. Becker and Catherine Mooney. *Categorizing compiler error messages with principal component analysis*. 12th China-Europe International Symposium on Software Engineering Education, Shenyang, China, May 2016.

7 Brett A. Becker. *A new metric to quantify repeated compiler errors for novice programmers* Proceedings of the 21st Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2016), Arequipa, Peru, July 2016. ACM. DOI: 10.1145/2899415.2899463

8 Brett A. Becker. *An exploration of the effects of enhanced compiler error messages for computer programming novices*. MA Thesis, Dublin Institute of Technology, Dublin, Ireland, November 2015.

9 Paul Denny, Andrew Luxton-Reilly and Dave Carpenter. *Enhancing syntax error messages appears ineffectual*. In Proceedings of the 19th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2014), Arequipa, Peru, July 2014. ACM. DOI: 10.1145/2591708.2591748

10 Devon Harker. *Examining the effects of enhanced compilers on student productivity*. MSc Thesis, University of Northern British Columbia, Prince George, British Columbia, Canada, 2018.

11 Michael J. Lee and Andy J. Ko. *Personifying programming tool feedback improves novice programmers learning*. In Proceedings of the 7th International Workshop on Computing Education Research (ICER 2011), Seattle, Washington, USA, August 2011, ACM. DOI: 10.1145/2016911.2016934

12 Raymond S. Pettit, John Homer and Roger Gee. *Do enhanced compiler error messages help students?: Results inconclusive*. Proceedings of the 47th ACM Technical Symposium on Computer Science Education (SIGCSE 2017), Seattle, Washington, USA, March 2017, ACM. DOI: 10.1145/3017680.3017768

13 James Prather, Raymond Pettit, Kayla Holcomb McMurray, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. *On novices' interaction with compiler error messages: A human factors approach*. Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER 2017), Tacoma, Washington, USA, August 2018, ACM. DOI: 10.1145/3105726.3106169

3.5 Visualizing and Interpreting Multimodal Data

Tanja Blascheck (INRIA Saclay – Orsay, FR)

Main reference Tanja Blascheck, Kuno Kurzhals, Michael Raschke, Michael Burch, Daniel Weiskopf, Thomas Ertl: "Visualization of Eye Tracking Data: A Taxonomy and Survey", Comput. Graph. Forum, Vol. 36(8), pp. 260–284, 2017.
URL http://dx.doi.org/10.1111/cgf.13079

In program language design, evaluation plays a crucial role. In recent years, eye tracking has become one means to analyze how participants perceive and understand a programming language. Because programming is highly interactive, a study should take interaction into account as well. In addition, think-aloud data gives insights into cognitive processes of participants using a programming language. Typically, researchers evaluate these data sources separately. However, it is beneficial to correlate eye tracking, interaction, and think aloud data for deeper analyses. I present challenges and possible solutions in triangulating behavior using multiple evaluation data sources and how these approaches can be used to analyze programming languages. Overall, the objective of this talk is to provide methods and techniques that contribute to more holistic evaluation methods that, in turn, leads to an improved understanding of programming languages. Starting by exploring existing evaluation methodologies, a description of the state-of-the-art of visualization techniques [5] that are available for studying eye movement data are provided and it is discussed, to what extent the existing techniques can be applied. To integrate the use of eye movement data with log files and think-aloud protocols, three novel visualization techniques are contributed: the Radial Transition Graph [6], the AOI Sequence Chart [1], the AOI hierarchy approach [2], and a Visual Coding Approach [3], which embeds Word-sized Eye Tracking Visualizations [4].

References

1 T. Blascheck, M. John, K. Kurzhals, S. Koch, and T. Ertl. *VA²: A visual analytics approach for evaluating visual analytics applications*. IEEE Transactions on Visualization and Computer Graphics, 22(1): 61–70, 2016.
2 T. Blascheck, K. Kurzhals, M. Raschke, S. Strohmaier, D. Weiskopf, and T. Ertl. *AOI hierarchies for visual exploration of fixation sequences*. In Proceedings of the Symposium on Eye Tracking Research & Applications, pages 111–118. ACM, 2016.
3 T. Blascheck, F. Beck, S. Baltes, T. Ertl, and D. Weiskopf. *Visual analysis and coding of data-rich user behavior*. In Proceedings of the IEEE Conference on Visual Analytics Science and Technology, pages 141–150. IEEE Computer Society Press, 2016.
4 F. Beck, T. Blascheck, T. Ertl, and D. Weiskopf. *Word-sized eye tracking visualizations*. In Eye Tracking and Visualization, pages 113–128. Springer, 2017.
5 T. Blascheck, K. Kurzhals, M. Raschke, M. Burch, D. Weiskopf, and T. Ertl. *Visualization of eye tracking data: A taxonomy and survey*. Computer Graphics Forum, 2017.
6 T. Blascheck, M. Schweizer, F. Beck, and T. Ertl. *Visual comparison of eye movement patterns*. Computer Graphics Forum, 36(3): 87–97, 2017.

3.6 Methodological Considerations

Brian Dorn (University of Nebraska – Omaha, US) and Briana B. Morrison (University of Nebraska – Omaha, US)

In this talk we review conceptual and practical elements of empirical study design. Multiple epistemological stances on evidence are introduced, and we frame the work of programming as activity to be understood holistically using the activity theory framework. We then discuss specific methods and tools from computer science education and the learning sciences that may be useful to the programming language design community when designing studies.

3.7 I Don't Understand Program Comprehension

Johannes C. Hofmeister

How do we understand programs? When we ask this question, it is worthwhile to think about what comprehension is and how we can make it measurable. If we don't, we might end up asking the wrong questions. For example, several studies had asked how syntax highlighting affects comprehension. But why should recall questions be answered differently depending on the color of the words? I proposed that highlighting does not affect memory and reasoning, and instead affects perceptual processes. This idea was demonstrated visually. The talk showed how psychologists reason about such ideas, by introducing explaining variables, named constructs.

3.8 What evidence do we have about programming language design?

Antti-Juhani Kaijanaho (University of Jyväskylä, FI) and Andrew J. Ko (University of Washington – Seattle, US)

In this talk we present an informal review of the body of evidence about programming language design. The talk discusses the results of two mapping studies of empirical studies about programming languages, and a literature review of evidence about the effect of programming languages on productivity and errors. The talk concludes that we don't know very much (there are only about 100 studies evaluating language features), that the number of studies is accelerating, but very slowly, and that most studies focus on evaluating novel language features, rather than systematically comparing language features in broad use. The researchers doing these studies tend to be trained in labs with expertise on running human subjects experiments.

3.9 Scientists Programming

Amelia A. McNamara (Smith College – Northampton, US)

Main reference Julia S. Stewart Lowndes, Benjamin D. Best, Courtney Scarborough, Jamie C. Afflerbach, Melanie R. Frazier, Casey C. O'Hara, Ning Jiang, and Benjamin S. Halpern: "Our path to better science in less time using open data science tools." Nature Ecology & Evolution, 1, 2017
URL https://doi.org/10.1038/s41559-017-0160

Scientists are an important important sub-category of programmers, because their work has implications in government policy, medicine and general knowledge. The three main categories of computer use by scientists are "make models, generate data" "generate data, make models," and general application programming. We will not consider the last category in this talk, as it is the closest to traditional software engineering. Common tools for making models to generate data are Matlab, Sage, Maple, and Mathematica. Scientists who generate data (by field collection, experimentation, etc) and make models, often use Excel, Stata, SPSS, SAS, R or Python. The transition from non-reproducible research to more robust analysis can be challenging, but the benefits are broad. We consider as a case study a marine science group who wrote "Our path to better science in less time using open data science tools" about their movement from Excel to a toolkit including git and GitHub for collaboration and version control, R in the IDE RStudio for analysis, and RMarkdown for reproducibility and narrative.

3.10 Types of Studies

Brad A. Myers (Carnegie Mellon University – Pittsburgh, US), Jonathan Aldrich (Carnegie Mellon University – Pittsburgh, US), Michael Coblenz (Carnegie Mellon University – Pittsburgh, US), and Joshua Sunshine (Carnegie Mellon University – Pittsburgh, US)

Main reference Brad A. Myers, Andrew J. Ko, Thomas D. LaToza, YoungSeok Yoon: "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools", IEEE Computer, Vol. 49(7), pp. 44–52, 2016.
URL http://dx.doi.org/10.1109/MC.2016.200

Traditional programming language design approaches center around theoretical and performance-oriented evaluation. Recently, researchers have been considering more approaches to language design, including the use of quantitative and qualitative user studies, to evaluate how different designs affect users. In this talk, we define "design" to encompass both the activities before and during creation of a system, including needs finding, and the activities involved in the evaluation after the design has been created. We list desiderata of programming languages, including software engineering quality attributes such as correctness, performance of the resulting code, expressiveness, speed of compiling, understandability, ease of reasoning, modifiability and learnability. We identify the different kinds of people or roles involved, such as logician, industrialist, empiricist and teacher. Then, we list many different methods that can be used to help with the design, most of which we have used in our group. These include: contextual inquiry, interviews, surveys, corpus studies, natural programming, rapid prototyping, programming language and software engineering theory, qualitative usability studies, case studies, expert evaluation, performance evaluation, user experiments, and

formalism and proof. We argue for taking an interdisciplinary approach using mixed methods to answer questions. Finally, we argue against having unsubstantiated claims, using methods incorrectly, assuming the conventional wisdom, and inadequate reporting of results.

3.11 Programming language cultures are relevant

Lutz Prechelt (FU Berlin, DE)

Main reference Lutz Prechelt: "Plat_Forms: A Web Development Platform Comparison by an Exploratory Experiment Searching for Emergent Platform Properties", IEEE Trans. Software Eng., Vol. 37(1), pp. 95–108, 2011.
URL http://dx.doi.org/10.1109/TSE.2010.22

We will contrast glimpses of possible cultures (found in self-presentations) of different programming languages: Haskell vs. C++ vs. Go; PHP vs. Ruby; Perl vs. Python. These suggest that distinct cultures may exist for at least many languages. Then we discuss two pieces of actual evidence that such cultures have relevant impact: First, in one experiment the Java participants all chose either a simple, very inefficient or a super-efficient, but very cumbersome solution, while the Perl and Python participants chose a third one, based on HashMaps, that is simple *and* efficient. Second, in another experiment 4 of 6 Perl and PHP teams chose the most straightforward and by far most maintenance-friendly approach, while the Java teams picked a conventional one involving property files, although that is cumbersome and its only advantage (internationalization) was explicitly not required.

References

1 Lutz Prechelt. *An empirical comparison of seven programming languages*. IEEE Computer 33(10), October 2000.
2 Lutz Prechelt. *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program*. Technical Report 2000-5, 34 pages, Fakultät für Informatik, Universität Karlsruhe, Germany, March 2000.
3 Lutz Prechelt. *Plat_Forms: A Web Development Platform Comparison by an Exploratory Experiment Searching for Emergent Platform properties*. IEEE Trans. on Software Engineering 37(1):95-108, Jan-Feb 2011.
4 Lutz Prechelt. *Plat_Forms 2007: The Web Development Platform Comparison – Evaluation and Results*. Technical Report TR-B-07-10, 118 pages, Freie Universität Berlin, Institut für Informatik, June 2007.

3.12 Eye Tracking in Program Comprehension

Bonita Sharif (Youngstown State University, US)

Main reference Timothy R. Shaffer, Jenna L. Wise, Braden M. Walters, Sebastian C. Müller, Michael Falcone, Bonita Sharif: "iTrace: enabling eye tracking on software artifacts within the IDE to support software engineering tasks", in Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 – September 4, 2015, pp. 954–957, ACM, 2015.
URL http://dx.doi.org/10.1145/2786805.2803188

Eye tracking has been used since the 1800s for various tasks in psychological research. An eye tracker tells us what a person is looking at while they are doing a task. This information is invaluable for researchers and educators. Educators can see the thought processes that

go into reading and comprehending code. This can help them devise better intervention strategies. Researchers can see how developers work while fixing a bug or adding a feature and use this information to devise better methods and tools for them. In this talk, I will describe eye tracking technology, introduce three eye tracking studies related to program comprehension and describe how we can conduct empirical studies at scale using iTrace, a tool that implicitly embeds eye tracking into the developer work environment. In conclusion, a discussion about how these results could help PL designers and educators is presented.

References

1 Bonita Sharif, Timothy Shaffer, Jenna L. Wise and Jonathan I. Maletic. *Tracking Developers' Eyes in the IDE*. IEEE Software, 33:3, 2016
2 Bonita Sharif and Jonathan Maletic. *iTrace: Overcoming the Limitations of Short Code Examples in Eye Tracking Experiments*. IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2 – 7, 2016
3 Katja Kevic, Braden M. Walters, Timothy R. Shaffer, Bonita Sharif, David C. Shepherd and Thomas Fritz. *Tracing software developers' eyes and interactions for change tasks*, ESEC/FSE, Aug 30 – Sept 4, 2015
4 Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha E. Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif and Sascha Tamm. *Eye movements in code reading: relaxing the linear order*, IEEE International Conference on Program Comprehension, ICPC, Florence, Italy, May 16 – 24, 2015
5 Bonita Sharif and Jonathan Maletic. *An Eye Tracking Study on camelCase and under_score Identifier Styles.*. IEEE International Conference on Program Comprehension, ICPC, Braga, Portugal, USA, Jun 30 – Jul 2, 2010

3.13 Early Experimental Studies of Conditionals in Programming Languages

Walter F. Tichy (KIT – Karlsruher Institut für Technologie, DE)

Eight controlled experiments about conditionals in programming languages are surveyed. The experiments were identified in a mapping study by Kaijanaho as the only controlled studies on conditionals between 1973 and 1993. They evaluated the goto statement, arithmetic if, nested if-then-else, scoping keywords such as `begin` and `end`, decision tables, `if-then` rule sets, and indentation. The first paper, Sime et al's study of 1973, compared goto statements with nested if-then-else without scoping constructs. The experiment used a specially designed micro language that isolated the language feature under study, a technique that was taken up by later experiments. A 1977 study by the same authors added scoping and indentation, while a 1984 paper by Vessey et al used decision tables rather than text to specify the statements to be written. A 1978 paper by Embley advocated using empirical studies in conjunction with considerations of the complexity of the associated proof rules when designing new control constructs. Two studies compared rule sets with `if-then-else` and `goto`. Even the effect of the true/false direction of Boolean expressions in conditionals was tested.

From the standpoint of experimental methodology, these studies were well constructed, but they appear to have had little impact on programming language design. For example, the nested if-then-else statement was already present in the Algol programming language in 1960, and Dijkstra's famous letter "The Go To Statement Considered Harmful" (which called for the abolishment of the goto statement) appeared in 1968, years before Sime's controlled

trial of 1973. The surveyed papers pay scant attention to the intense debate about structured programming at the time, which started with Dijkstra's 1968 letter. Knuth published a thorough study of control constructs, including the goto and its alternatives, in 1974. By 1978, the C language provided an adequate set of control constructs, including `if-then-else`, curly braces instead of unwieldy keywords, short-circuit evaluation of Boolean expressions, as well as switch, continue, break, and return statements, which are constrained forms of goto. This set has stood the test of time and is still in use in more recent languages such as Java. Apparently, language designers, by personal programming experience, introspection, or intuition, found adequate forms of conditionals without recourse to controlled trials. However, not all questions about programming languages can be settled that way. For example, the questions of whether to prefer dynamic over static type-checking, or functional over imperative programming, need evidence to be answered reliably.

References

1. Edsger W. Dijkstra, Letters to the Editor: Go To Statement Considered Harmful, Communications of the ACM, 11(3), March 1968, 147-148. DOI: 10.1145/362929.362947
2. David W. Embley, Empirical and formal language design applied to a unified control construct for interactive computing, Int. J. Man-Machine Studies (1978) 10 (2), 197-216. DOI: 10.1016/S0020-7373(78)80012-1
3. D. J. Gilmore and T. R. G. Green, Comprehension and recall of miniature programs, Int. J Man-Machine Studies (1984), 21 (1), 31-48. DOI: 10.1016/S0020-7373(84)80037-1
4. R. Halverson, An Empirical Investigation Comparing IF-THEN Rules and Decision Tables for Programming Rule-Based Expert Systems, In System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on. Vol. iii, 316–323 vol.3.DOI: 10.1109/HICSS.1993.284327
5. E. R. Iselin, Conditional statements, looping constructs, and program comprehension: an experimental study. Int. J. Man-Machine Studies (1988) 28 (1), 45–66. DOI: 10.1016/S0020-7373(88)80052-X
6. A.-J. Kaijanaho, Evidence-Based Programming Language Design, A Philosophical and Methodological Exploration, Dissertation, Faculty of Information Technology of the University of Jyväskylä, Finland, 2015
7. B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall, 1978.
8. D. E. Knuth, Structured Programming with go to Statements, ACM Computing Surveys, 6(4), 1974, 261-301. DOI: 10.1145/356635.356640
9. P. Naur (ed.), Report on the Algorithmic Language Algol 60, Comm. of the ACM, 3(5), 1960, 299-314. DOI: 10.1145/367236.367262
10. B. Shneiderman, Exploratory experiments in programmer behavior. Int. J. Computer and Information Sciences (1976) 5 (2), 123–143. DOI:10.1007/BF00975629
11. B. Shneiderman, and R. Mayer, R. Syntactic/semantic interactions in programmer behavior: A model and experimental results. Int. J. Parallel Programming (1979) 8 (3), 219–238. DOI: 10.1007/BF00977789
12. M. E. Sime, T. R. G. Green, and D. J. Guest, Psychological evaluation of two conditional constructions used in computer languages. Int. J. Man-Machine Studies 5 (1), (1973) 105–113. DOI: 10.1016/S0020-7373(73) 8001-2
13. M. E. Sime, T. R. G. Green, and D. J. Guest. Scope marking in computer conditionals – a psychological evaluation. Int. J. Man-Machine Studies 9 (1), (1977) 107–118. DOI: 10.1016/S0020-7373(77)80045-X
14. I. Vessey and R. Weber, Conditional statements and program coding: an experimental evaluation. Int. J. of Man-Machine Studies 21 (2), (1984) 161–190. DOI: 10.1016/S0020-7373(84)80065-6

4 Future Research Questions and Studies

The seminar included a brainstorming session where the goal was to enumerate research questions. The session was free-form and questions that were mentioned or written down by participants are listed here. To be clear on the intent here, our goal with this section was to document many of the research questions mentioned by participants.

We categorized the questions into broader themes after the event. We did this for organization in this document and to make it easier to see the big picture of what participants were interested in, but participants themselves did not have these categories during the session. Finally, while we did conduct some light editing, and removed a few questions that did not feel fleshed out enough to include, we did not substantively edit the original questions.

4.1 Impacts in the field

These questions appeared related to either how programming languages evolved, or could evolve, in regard to their impact in practice. We imagine many other questions on the topic could be asked.

4.1.1 Standardization

Research Question What aspects of programming languages have been largely standardized?

Description Over the last 50 years, while there has been some disagreement across programming languages, it is clear that some features have been more successful than others. Which parts have been standardized across multiple languages, and why?

4.1.2 Library design

Research Question Different software developers, given a library or API design problem, make different design choices. What can we learn about programming language and API design from the diversity of choices made by experts?

4.1.3 Polyglot programming

Research Question What is the impact of polyglot programming? Are there economic costs of the current polyglot state?

Description Often programmers need to embed one language in another (e.g. embedded SQL, or JavaScript inside HTML inside Ruby). What impact does this have?

4.1.4 Methods of language evolution

Research Question Characterize the in-use methods of programming language evolution among designers and maintainers of existing languages. To what extent are these consistent or inconsistent? How do these relate to the costs and benefits of programmers adopting these changes and using the new language features? Can we predict the costs/benefits of language changes?

Maybe eventually a given language is “done” and the cost of change outweighs any benefit. What is the effect of language feature creep on development?

4.2 Methodology

There was interest amongst participants in the methodological procedures under which we can test languages. This might include trying to garner a better understanding who we are, or perhaps who we should test in such studies, in addition to asking questions about the kinds of metrics we might use for evaluation.

4.2.1 Representativeness of participants

Research Question How representative are the students or subjects we use now? Which groups of people are we using now? We could compare new groups to the current group. This was an issue in psychology as well [1] and in medicine [2]. This also includes the diversity of the stakeholders.

References

1 http://www.slate.com/articles/health_and_science/science/2013/05/weird_psychology_social_science_researchers_rely_too_much_on_western_college.html
2 http://www.theguardian.com/lifeandstyle/2015/apr/30/fda-clinical-trials-gender\-gap-epa-nih-institute-of-medicine-cardiovascular-disease

4.2.2 Measuring cognitive load

Research Question What empirical methods can be used to unobtrusively measure cognitive load? The goal would be measure and propose interventions that improve productivity.

Description We don't yet know how to measure cognitive load in programmers, but we suspect that programmers' ability to be productive decreases when cognitive load is too high. We should identify and standardize methods that can be used to (minimally-invasively) measure cognitive load. Then, we should show that cognitive load indeed interferes with programmer effectiveness. Finally, we should create effective interventions that help decrease cognitive load (or help people be more effective when it is high. Eye tracking methodology lends itself well for this type of analysis.).

4.3 Language features

Many programming languages have features in common. While some have unique philosophies or semantics, many have commonalities like loops, conditionals, functions, classes, or other features. Even if the underlying semantics are the same, or very similar, language designers often use alternative notations or representations for them. Other times, designers vary the semantics in subtle ways. A variety of questions were asked regarding individual language features or aspects of their paradigms.

4.3.1 Functional vs. procedural languages

Research Question How are people **reasoning differently in functional vs procedural languages**? Where are they spending their time? Can we characterize how they are working? How does this look different from the existing literature on procedural languages?

Study Type Exploratory Qualitative Work

Description How do we characterize how functional programmers work? We could study an expert functional programmer by giving them a design problem and asking them to solve it and implement a corresponding program. Asking participants to work in pairs could

enable capturing their thoughts better because they would need to communicate with each other. For comparison purposes, the study might have five functional teams, five object-oriented teams, and five procedural teams.

The design could include multiple problems that seem more biased towards one type of language or the other, enabling analysis regarding the effect of the language paradigm. Further research questions include:

- How can we predict the task/language fit for a given task and language pair?
- What does it mean to pick the right language for the task? Particularly, if a language supports more than one paradigm, how do developers choose a specific paradigm?

References

1 Amjad Altadmri and Neil C.C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, New York, NY, USA, 522-527. DOI: http://dx.doi.org/10.1145/2676723.2677258

4.3.2 Concurrency and parallelism

Research Question What are appropriate language representations for concurrency and/or parallelism?

Description How do people think about these kinds of programs? Perhaps understanding human cognition would help develop a more effective way of writing concurrent and/or parallel code.

4.3.3 Creativity vs. Utility

Research Question What are the best language features to support creativity and how do they differ from those that provide other kinds of benefits?

4.3.4 Making the average programmer much better

Research Question Perhaps current languages are still too low-level. How could we make the average programmer 10x better? Some ideas for language approaches include:

- Search-based
- Machine learning
- Programming by example and imitation
- Programming by question-answer
- Natural language

In databases, users moved from databases to analysis of big data. How would that look like for programming? We should not only think about languages, but also about tools/IDEs and processes.

4.4 Novices and learning

Programming languages are used at all levels, from Kindergarten in many U.S. schools to seasoned professionals working in the field. While the spectrum is broad, some participants were interested in learnability or ease of use by those earlier in a lifetime. These questions thus focused around issues they thought might be relevant.

4.4.1 Learning performance of CS concepts

Research Question How does language design affect performance when students are studying various topics in computer science?

4.4.2 Learnability

Research Question What factors affect learnability of individual language features?

4.4.3 Attributes of authenticity

Authenticity typically concerns the extent to which users of educational programming tools (such as block-based editors) feel that they are learning "authentic" programming (as opposed to learning a different set of skills that do not count as "real" programming). However, there is an argument that authenticity applies to professionals as well.

Research Question What are the principal attributes of authenticity in programming languages?

Description For example, how do we prevent authenticity questions from stifling innovation?

4.4.4 Accents: implications of training on cross-language use

Research Question What is the impact of one's first programming language on subsequent programming behavior? What is the impact on language tools on programmer behavior across language environments? For example, at code.org, users are given the blocks in advance, so users do not practice the skill of finding a block. There was a position paper about this at VL/HCC 2015: http://www.felienne.com/archives/4352

4.4.5 Error handling and learning

Research Question Does graceful error handling improve the experience of learning programming for novices?

4.4.6 Novice vs. Expert compiler usage

Research Question How do published novice programmer compilation behaviors compare to that of industry developers?

Description Replicate a set of Blackbox [1] studies against Google developer logs and quantitatively compare distribution of error rates and other statistics. This serves as a bridging question between academia and industry to move the field forward. What can we learn from novices to help experts? What can we learn from experts to help novices?

4.4.7 Error messages

Research Question How do users read error messages and how is it affected by different error message content or display choices?

4.4.8 Error-proneness

Research Question What is error-proneness?

Description What leads to incorrect thinking / logic errors? Come up with a formal definition that would allow this to be checked. All PL designers want to have low values for error-proneness. Ideally come up with something that could be objectively measured.

4.5 Cognition and affect

A variety of questions were asked that were related to users thinking or feeling about programming. Such attributes could inform us about reasons for adoption or other factors of interest.

4.5.1 Thoughts about program behavior

Research Question How do people think about what a program does?

Description Try to evaluate program understanding. Is there an analogy to "code-switching" from linguistics? If so, is code switching good or bad? What models should we have of programming and how do we apply them to writing code? How do models change during programming? What's the relationship to cognitive load and working memory?

4.5.2 Language Love

Research Question What makes people love a language?

Description Evaluate what factors influence people's feelings about languages. Is loving programming not at all related to loving a programming language? Maybe culture and the ecosystem matter too.

4.5.3 Programmer descriptions of structures

Research Question In what ways do programmers describe the structure of their systems? And how do such descriptions help with programming tasks?

Description *Study 1*: Interview developers to see how they describe system structures. Ask them to describe the system in several different ways; how do those descriptions differ, and what order are they described in? Watch them drawing diagrams and talking through them in a think-aloud study.

Study 2: How do these descriptions prime a programmer for completing tasks? Prompt half of the participants to describe a system in one way prior to completing a task. The other half of the participants would just get the task without having to describe the system first. Measure productivity and interaction (possibly using eye tracking).

4.5.4 Mental spatial structures

Research Question Learn what mental spatial structures people develop regarding code (and what affects this). For example, how do people learn where (in a given file) particular pieces of code are? How does this differ between people? Is this different for people with various disabilities?

5 Working groups

The participants divided into working groups to focus on some of the research questions above that attracted the most interest and had potential for significant contributions.

5.1 Novices: polyglot questions

Johannes Bechberger (Karlsruher Institut für Technologie, DE), Scott Fleming (University of Memphis – Memphis, US), Briana B. Morrison (University of Nebraska – Omaha, US), Bonita Sharif (Youngstown State University – Youngstown, US), Andreas Stefik (University of Nevada, Las Vegas – Las Vegas, US)

This working group focused on questions on *polyglot* programming, which regards working in multiple programming languages at once. What happens when a programmer learns multiple languages? To what extent does knowledge transfer between languages? Is there an impact on productivity for domains, like the web, where programmers are forced to use multiple languages at once?

Study participants could be evaluated at several times for a longitudinal study: for example, after two semesters, after the end of the semester in which they learn their second language, and at graduation. At each time, they might be given code for various tasks in multiple languages and ask them to explain the code, even though they haven't necessarily seen those languages before. They could also be asked to trace the execution with various inputs. Eye tracking might provide insight into the participants' approach to the problem and help identify whether the participants recognize beacons [1], which help programmers understand the structure and behavior of programs. It might be helpful to do the study on pairs of participants rather than individual participants so that experimenters can observe the conversation. Regarding language learning, the group also discussed the question of whether it is better to learn one language deeply or learn multiple languages at once.

References

1 Susan Wiedenbeck. Beacons in computer program comprehension. International Journal of Man-Machine Studies 25.6 (1986): 697-709.

5.2 Context Switching and Cognitive Load

Ameer Armaly (University of Notre Dame – Notre Dame, US), Andrew Begel (Microsoft Research – Redmond, US), Igor Crk (Southern Illinois University – Edwardsville, US), Tanja Blascheck (INRIA Saclay – Orsay, FR), John Daughtry (Google Inc. – Seattle, US), Rob DeLine (Microsoft Corporation – Redmond, US), Ciera Jaspan (Google Inc. – Mountain View, US) , Philip Merlin Uesbeck (University of Nevada, Las Vegas – Las Vegas, US)

This working group focused on enumerating interesting research questions. Part of the conversation overlapped with the *Novices* group, with a shared interest in polyglot programming. The idea is that a polyglot programming scenario might lead to higher cognitive load, as programmers must remember information pertaining to multiple (potentially inconsistent) programming methodologies. Language designs that facilitate polyglot programming might involve addressing language inconsistencies, such as in loop constructs; missing constructs (one could compare languages in which one has a construct that the other lacks); and embeddings, in which programmers could embed code in one language inside code written in another language.

The group discussed the need for better methodologies to measure cognitive load; to be usable for a programming study, the methods must be non-destructive and non-interfering. Low cost is also important for practical reasons. The group was interested in how programming language designs and program design decisions correlate with cognitive load. Do particular

programming language constructs correlate with cognitive load? The group hypothesized that the number of types in use correlates with cognitive load, and that cognitive load might correlate with typos or stutter. The current results regarding cognitive load seem to try to categorize users as being overloaded or not rather than trying to assess cognitive load in a quantitative way. It might be interesting to consider the units of cognitive load so that it can be correlated more directly.

5.3 Language features and error-proneness

Jonathan Aldrich (Carnegie Mellon University – Pittsburgh, US), Michael Coblenz (Carnegie Mellon University – Pittsburgh, US), Andrew J. Ko (University of Washington – Seattle, US), Thomas LaToza (George Mason University – Fairfax, US), Sibylle Schupp (TU Hamburg-Harburg, DE), Walter Tichy (KIT – Karlsruher Institut für Technologie, DE)

Programming language designers tend to think a lot about user errors (bugs) because a lot of the techniques used in language design are intended to exclude particular classes of bugs. As a result, the working group on *language features* focused on creating a *grand theory of error-proneness of language features*, specifically regarding errors that emerge through reasoning, not perceptual slips or conflicts with prior knowledge.

Language features are abstractions. As such, they hide certain aspects of execution for the benefit of simpler reasoning. For example, consider integer division, which hides remainders, integer truncation, and division by zero. If, in hiding that complexity, the language feature allows the developer to never have to reason about that internal complexity, error-proneness is low. However, if the language feature still occasionally forces the developer to have to reason about that internal complexity, that reasoning will be even harder because the complexity is hidden, and therefore error-proneness may be higher. For example, 10/3 requires a programmer to reason about integers versus floating-point numbers. 10/0 requires programmers to reason about runtime errors.

The group hypothesized that the mechanism of error production is that developers need to be able to reason correctly about the behavior of an abstraction. If they do not have access (e.g. via training) to a correct model of that behavior, they will make mistakes. Abstractions occasionally hide behavior developers must reason about. Furthermore, error-proneness may arise by composition of language features: language features may interact with each other in ways that are error-prone.

5.3.1 Examples

- The division operator in most languages fully encapsulates the complexities of division, but not in the case of 0 or floating-point. Those nuances are not visible in the / operator in most languages, reducing the visibility of the need to handle those cases, increasing the likelihood of errors.
- Constraints make it easier to express declarative properties between values, but one must understand the hidden semantics of constraint satisfaction algorithms to avoid unintended side effects of cycles.
- Memory-safe languages allow programmers to think in terms of abstract objects and fields instead of the linear memory on which those objects are imposed. This abstraction eliminates many non-local interactions (or alternatively, "safety rules") that programmers have to consider in unsafe languages. The abstraction rarely breaks down in terms of

correctness; the main cases where it does are interaction with code written in unsafe languages (e.g. the native code interface in Java). It does break down in cases where an application's performance or memory-usage needs are strict enough to be affected by the costs of the garbage collection and dynamic checks necessary for memory safety.
- Monads (are these error-prone or hard to use? Are those different?)

5.3.2 Operationalization

Error-proneness is defined as semantics of an abstraction that are hidden but a developer must reason about correctly in order to be used correctly. Error-proneness may have a trade off with expressiveness because increased expressiveness requires abstraction, which hides complexity. If one must reason about that abstraction, an error might occur if that abstraction is hard to understand or use. This is related to the idea of leaky abstractions, but that work focuses more on architectural consequences and not language features o not error-proneness.

The theory might be used as a thinking tool in a language design process, and as a way to generate hypotheses to test. Doing so might validate the theory: if it effectively predicts parts of designs that are error-prone, it is a useful theory. APIs might also be analyzable in the same framework. The group also considered perceptual sources of errors from syntax; these are more akin to *slips* than to *mistakes*, depending on whether one considers these terms to be substantively different, but they are errors nonetheless.

5.3.3 Evaluation

Researchers might work to test the theory by building a predictive error model and showing that it makes useful predictions for particular language features. Future work should figure out criteria for thresholds: how much complexity is enough to cause error-proneness? What does this depend on: experience, etc...? In the future, researchers should compare fine-grained differences in what an abstraction hides and compare developers' defect production.

5.4 Why do people love programming languages?

Igor Crk (Southern Illinois Univ. – Edwardsville, US),Andrew Duchowski (Clemson University, USA), Matthias Hauswirth (University of Lugano, Switzerland), Brad A. Myers (Carnegie Mellon University – Pittsburgh, US), Craig Anslow (Victoria University of Wellington), Brian Dorn (University of Nebraska, Omaha), Lutz Prechelt (FU Berlin, Germany), Antti-Juhani Kaijanaho (University of Jyväskylä, Finland)

This working group focused on understanding *affect*: what is it about languages that causes people to love or hate them? Is it primarily due to technical attributes (e.g. features) or is personal relevance more important? For example, people might love the first language they used, or their feelings might be driven by their prior positive (or negative) experiences using languages on particular projects.

The group designed an interview study in which participants would be asked:
- Is there a language you love, and why?
- Tell me about a time when a language feature helped you achieve a goal.
- Tell me about a time when a language feature stood in the way of a goal.
- Tell me about a time when you abandoned a language in favor of another one.

In the discussion, the participants noted that it is easy to conflate love with adoption. Sometimes people don't have choices because their language selection is dictated by their employer or other practical considerations. Any study of this would need to separate the questions of adoption from the questions of emotional attachment, and separate the influences of the various factors. The study might leverage standard business or psychological measures of affect.

The group was interested in what qualities of people cause them to be drawn to particular languages. One might ask: "How would one characterize people who love language X?" The group was also interested in the progression of feelings over time as people learn more languages (for example, at a university). By asking senior programmers, teachers, and students at various stages, one could study how language learning affects feelings about languages. It might also be interesting to compare adoption and use to what people love. Other study techniques might involve asking programmers who know at least two languages well for comparisons and explanations of what languages they do not like. It is important to include social aspects in this discussion; perhaps people like or dislike languages in part because of the people in the language communities. Perhaps some people love programming languages in general rather than loving specific languages. Another question the group identified pertained to global adoption: is there a map of which languages are in use in each part of the world? If not, is it worth creating?

5.5 Representativeness of subjects in studies

Felienne Hermans (TU Delft, NL), Johannes Hofmeister (Universität Passau, DE), Amelia A. McNamara (Smith College – Northampton, US), Lea Verou (MIT – Cambridge, US)

This working group discussed a practical problem that the community has encountered when running studies: most studies so far have been on university students at high-quality universities who have studied some programming. This biases the sample toward white, male, able-bodied, high socio-economic status students. The group identified three mechanisms to mitigate the impact of this problem:

1. Studies should be reporting gender, SES, ethnicity. However, the European Union has restrictions on what data can be collected, which may make tracking ethnicity difficult for researchers in the EU.
2. Run studies on other groups to assess impact on alternative demographics. Medicine has had this problem too: men were used largely in the 1940s because of the connections to the military in early controlled experiments, for example.
3. Make sure we understand the results from psychology and HCI in this area [1].

References

1 Ari Schlesinger, Keith Edwards, Rebecca E. Grinter. (2017). Intersectional HCI: Engaging Identity through Gender, Race, and Class. 5412-5427.
https://doi.org/10.1145/3025453.3025766

6 Open problems

6.1 Evidence standards

The participants had a plenary discussion about the need for an evidence standard in human factors research in computer science. There seemed to be general consensus that an evidence standard would clarify to authors and reviewers how to report on their research to improve replication and to clarify the limitations of particular studies. The group discussed the details of the CONSORT evidence standard [1]. Some participants expressed concern at having to comply with a standard, but the strong majority seemed to find the idea reasonable. The group also had broad consensus that the types of information that the CONSORT standard requires in its reporting are a reasonable place to start when proposing an evidence standard.

6.1.1 Adoption of evidence standards

Some ideas that were proposed to assist **adoption** include:

- Send a draft of an evidence standard for publication. Where to send such work was unclear.
- Two approaches were considered for adoption of such a standard. One way was to use a top-down approach where we attempt to convince a publication venue to require its use for some papers. Another way would be to propagate the standard bottom-up by submitting papers that follow it, citing in each work in a consistent way. Both approaches have pros and cons.
- It is important to start with the right community. The IEEE VL/HCC conference (http://www.vlhcc.org) might be a good place to start because lots of the VL/HCC leadership are among the meeting participants. As such, perhaps a recommendation could be attached to the VL/HCC call for papers. In comparison, CHI has an informal evidence standard based on psychology papers. Alternatively, the computing education conferences and journals already use empirical data heavily, perhaps especially TOCE and ICER, and as such might be good targets.
- An evidence-based track at a conference might be a lightweight way to further adoption. Perhaps eventually the whole conference would be in this track.
- Make reviewers feel like they are the "last ones to know" about the evidence standard so that they feel like they should adopt the new standards of the "in" group.
- Change is not an instant process and working toward a standard is a marathon, not a sprint.
- The Open Science Framework is an existing mechanism to manage scientific workflows, including study registration. Study registration is a well-accepted technique in science to reduce a number of potential problems during the publication process (e.g., Type I error, fraud, "surprise factor"). Perhaps leveraging existing mechanisms could reduce the cost and difficulty of adoption.
- While CONSORT, the What Works Clearinghouse, APA, and other standards exist, it was unclear what evidence standard would work best for computer science. Given other disciplines already have such standards, we may be able to learn from them in designing ours, while potentially improving them to avoid historical mistakes.
- If an evidence standard were to be adopted at particular conferences or journals, or across conferences and journals, training will be needed to help scholars use and understand it, both at the reviewing and at the writing stages.

6.1.2 Criteria

Participants discussed some criteria for good evidence standards:

- Evidence standards should be clear so that authors know what the expectations are, but the standard should not unnecessarily make papers longer since page limits are of concern.
- Care should be taken to avoid unnecessarily rejecting papers to which the standards don't align well.
- The standard should include guidelines **both** for authors and reviewers.
- It needs to be possible to cite the evidence guidelines. Also it would be nice to be able to cite study guidelines.
- It should be clear that the evidence standard does not apply to all kinds of research. For example, there was a backlash in the UIST community when it became expected that authors include an A/B study of their work because some authors felt that this was inappropriate for their work.

References

1 Schulz, K.F., Altman, D.G., Moher, D., for the CONSORT Group. CONSORT 2010 Statement: updated guidelines for reporting parallel group randomised trials. Annals of Internal Medicine. 2010;152. Epub 24 March. http://annals.org/article.aspx?articleid=745807

Participants

- Jonathan Aldrich
Carnegie Mellon University – Pittsburgh, US
- Craig Anslow
Victoria University – Wellington, NZ
- Ameer Armaly
University of Notre Dame, US
- Johannes Bechberger
KIT – Karlsruher Institut für Technologie, DE
- Brett A. Becker
University College Dublin, IE
- Andrew Begel
Microsoft Research – Redmond, US
- Tanja Blascheck
INRIA Saclay – Orsay, FR
- Neil C. C. Brown
King's College London, GB
- Michael Coblenz
Carnegie Mellon University – Pittsburgh, US
- Igor Crk
Southern Illinois Univ. – Edwardsville, US
- John M. Daughtry
Google – Seattle, US
- Fabian Deitelhoff
Fachhochschule Dortmund, DE
- Rob DeLine
Microsoft Corporation – Redmond, US
- Brian Dorn
University of Nebraska – Omaha, US
- Andrew Duchowski
Clemson University, US
- Scott Fleming
University of Memphis, US
- Baker Franke
Code.org – Seattle, US
- Reiner Hähnle
TU Darmstadt, DE
- Matthias Hauswirth
University of Lugano, CH
- Felienne Hermans
TU Delft, NL
- Johannes Hofmeister
Universität Passau, DE
- Ciera Jaspan
Google Inc. – Mountain View, US
- Antti-Juhani Kaijanaho
University of Jyväskylä, FI
- Andrew J. Ko
University of Washington – Seattle, US
- Thomas LaToza
George Mason University – Fairfax, US
- Andrew Macvean
Google – Seattle, US
- Jonathan I. Maletic
Kent State University, US
- Amelia A. McNamara
Smith College – Northampton, US
- Briana B. Morrison
University of Nebraska – Omaha, US
- Brad A. Myers
Carnegie Mellon University – Pittsburgh, US
- Lutz Prechelt
FU Berlin, DE
- Sibylle Schupp
TU Hamburg-Harburg, DE
- Bonita Sharif
Youngstown State University, US
- Andreas Stefik
Univ. of Nevada – Las Vegas, US
- Walter F. Tichy
KIT – Karlsruher Institut für Technologie, DE
- Phillip Merlin Uesbeck
Univ. of Nevada – Las Vegas, US
- Lea Verou
MIT – Cambridge, US