2009

The Past, Present and Future of Programming in HCI

Brad Myers
Carnegie Mellon University

Andrew Ko
Carnegie Mellon University

The Past, Present and Future of Programming in HCI

Brad A. Myers
Human-Computer Interaction Institute, Carnegie Mellon University
Pittsburgh, PA 15213 USA
bam@cs.cmu.edu

Andrew J. Ko
The Information School, University of Washington
Seattle, WA
ajko@u.washington.edu

ABSTRACT

The first computer users were all programmers, and the field of Human-Computer Interaction started, in part, with a focus on improving how programming was done. There was a significant amount of work in the 1980's on this topic, but it mostly died out in the 1990s. Now, there is a resurgence of work on what used to be called the Psychology of Programming, Software Psychology, and the Empirical Studies of Programming. Now, research that combines HCI and software engineering concerns regularly wins awards at both the software engineering and HCI conferences, and although there is no longer a conference devoted solely to this topic, it is a major focus of the popular VL/HCC conference series. In this paper, we argue that new HCI and software engineering methods and tools, along with a new acceptance of the programming community, makes it a propitious time for a renewed focus on this topic.

EXTENDED ABSTRACT

One way to define "programming" is as the process of transforming a mental plan of desired actions for a computer into a representation that can be understood by the computer [10] Expressed this way, it seems obvious that the study of humans and programming should be a topic of HCI. Indeed, this area of study has a long history, and has many names, including the *Psychology of Programming* [30, 5, 10], *Software Psychology* [24], and *Empirical Studies of Programming*, which is also the name of a series of workshops from 1986-1999.

Most of the early work focused on studying professional programmers or novice programmers. A "professional" programmer might be defined as someone whose primary job function is to write or maintain software. A "novice" programmer might be defined as someone who is learning how to program. Recently, there has also been a focus on the category we are calling end-user programmers (EUP) [23] (also called *end-user developers*), who are people who write programs, but *not* as their primary job function. Instead, they write programs in support of achieving their main goal, which may be accounting, web design, office work, research, entertainment, etc. End-user programmers generally use special-purpose languages such as spreadsheet languages or web authoring scripts, but some EUPs, such as chemists or other scientists, may need to learn to use programming languages such as C or Java to achieve their programming goals.

In this talk, we briefly review the themes and results of the early work, and show how the current approaches are significantly different. We also discuss how many themes in HCI are converging on problems of customization and end-user programming, requiring new thinking about how to support individuals' unique programming requirements.

EARLY THEMES

Early research focused on how to make programming easier to learn for novices (see [21] for a survey). For example, many studies highlighted the problems that novices had, including syntax issues [6] , differences between programming languages and English [15], and theories on how to teach programming better (e.g., by teaching via "schemas" [27]).

Another theme of the early work was using graphical or visual programming (see e.g., [18, 25] for surveys). There was a widespread tendency to expect visual languages to be superior to text for novice programming, since two-dimensional visual perception might be more natural and efficient than reading, and visual programming environments reduce the need to rely on syntax. However, research showed that visual languages are not more natural than text and use screen space inefficiently [20]. Visual languages also have a high "viscosity," requiring effort in layout rearrangement when making changes [7].

Visualization was widely viewed as useful for helping people understand their programs and algorithms. Early systems visualized data structures [17], algorithms [2, 28], and executions [1]. However, subsequent research showed that visualizations often did not help with learning or understanding programs (this does not apply to scientific and information visualizations, which have a long history of making vast amounts of information understandable [3]), and was most useful when the student constructed their own visualizations [11].

Tools were created to specifically help with learning to program, including special-purpose languages for novices like Logo and Pascal. Syntax-directed editors (such as MacGnome [15], that help with the construction of textual languages by helping avoid the problems of syntax, were shown to help novices construct programs from a blank screen more quickly. However, like visual programs, they have high viscosity and make it more difficult to edit programs.

CURRENT THEMES

While similar work continued in the nineties, it occurred at a slower pace. Teachers for elementary-school children found little carry-over from programming to other topics, and other techniques failed to make programming much easier (as described above). Professional developers seemed to have an aversion to using tools that researchers developed.

Perhaps the biggest shift was driven by the introduction of new methodologies into software engineering conferences, in particular, the idea of directly observing the work of software engineers. One of the earliest papers marking this shift include Perry et al.'s study of communication among software developers [22], which was one of the first to find that software development work, despite stereotypes, actually involved considerable communication and cooperation. This thread of studies continued at conferences such as CSCW and GROUP through the early 2000's and continues today.

Many of the early work on software development *tools* was not useful (or at least not used) by professional developers, but in early 2000's, software engineering researchers started to take a more human-centered approach to the design and evaluation of these tools. For example, several tools designed by Murphy and students [16, 4] were explicitly motivated by studies of software developers' work difficulties. The same was true of our recent work on debugging [13]. The common themes among these and similar examples is that studies of software development inform design, and evaluations of designs inform further study. Furthermore, rather than focusing on technological

Brad A. Myers and Andrew J. Ko, "The Past, Present and Future of Programming in HCI". *Human-Computer Interaction Consortium* (HCIC'09) <http://www.hcic.org/hcic2009/>, Winter Park, CO. February 4th - 8th, 2009.
http://www.cs.cmu.edu/~natprog/papers/MyersKoHCIC09.pdf

novelties, the most respected of software development tool contributions focus on the questions [26] and information needs [12] fundamental to software development work.

A number of technological shifts have made many of these contributions feasible. For example, the Eclipse environment, developed by IBM, has been a catalyst in reinvigorating research on software development tools, since Eclipse allows researchers to focus on what they want to innovate, while providing the rest of the features that programmers require. The significant, long-term industrial backing of languages like Java and C# have also been instrumental, especially with these languages themselves having features useful for tools such as reflection, Java's instrumentation and recording framework, the Java Platform Debugger Architecture (JPDA), etc.

Furthermore, today's developers are much more likely to use an integrated development environment, rather than command line tools. This is due in part to the sophistication of these tools, along with an increased focus on programmer productivity, due, in part, to the outsourcing of programming jobs. Another factor is the explosion of open source development projects, which has furthered the development of development tools for collaborating asynchronously and remotely.

As these changes have occurred, work on understanding and supporting *end-user programming* has matured considerably. Beyond just research on new languages, this work has explored dozens of distinct populations of people who program to support their work, it has analyzed gender differences in software development tool use and adoption in end-user programming environments, and it has also developed a number of unique tools for increasing the correctness of end users' programs [19]. Many of these have been transitioned into more general software tools that professionals use.

FUTURE THEMES

While programming used to be at the center of HCI and is now splintered among a number of other disciplines, it is making its return to the forefront of HCI and HCI research. The rapid growth of blogs and social networking sites has led to an immense demand for customization, exposing millions of Internet users to snippets of HTML and Javascript. The proliferation of wikis has exposed the broader public to syntax issues. A major theme of intelligent interfaces is how to reveal the underlying program learned by a machine based on user feedback, without requiring that the users learn to program [29]. The growth of research on assistive technologies is demanding a closer look at how to support customization of more than just parameters. Ubiquitous computing is beginning to struggle with how to make sensor-based applications relevant to users without users having to learn some programming [8]. All of these trends are converging on the need for a better understanding of how to design and support programming (or at least programming-like functionality) that does not use conventional software engineering methodologies or failed approaches of the past.

REFERENCES

[1] Baecker R., DiGiano C. & Marcus A. Software Visualization for Debugging. *CACM*, **40**(4). 44-54, 1997.

[2] Brown M. & Sedgewick R. A System for Algorithm Animation, *Computer Graphics*, SIGGRAPH'84. 1984. 177-186.

[3] Card S.K., Mackinlay JD & Shneiderman B. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999.

[4] Cubranic & Murphy G. Hipikat: Recommending Pertinent Software Development Artifacts, *ICSE*, 2003.

[5] Curtis B. Fifteen Years of Psychology in Software Engineering: Individual Differences & Cognitive Science. *ICSE* 1984, 97-106.

[6] M.J. Fitter & T.R.G. Green. When Do Diagrams Make Good Computer Languages? *IJMMS*. 1979. **11** 235-261.

[7] Green TRG & Petre M. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework, *JVLC* 1996. **7**(2). 131-174.

[8] Hartmann B, Abdulla L, Mittal M & Klemmer SR. Authoring Sensor Based Interactions Through Direct Manipulation and Pattern Matching, *CHI*, 2007.

[9] Hoc JM, Green TRG, Samurçay R & Gilmore DJ, Eds. *Psychology of Programming*. London, Academic Press. 1990a.

[10] Hoc JM & Nguyen-Xuan A. Language Semantics, Mental Models and Analogy. *Psychology of Programming*. J.-M. Hoc, T. R. G. Green, R. Samurçay & D. J. Gilmore, Eds. 1990b: London, Academic Press. 139-156.

[11] C. Kehoe, J. Stasko & A. Taylor. Rethinking the evaluation of algorithm animations as learning aids: An observational study, *IJHCS*, **54**(2). 265-284, 2001.

[12] A. J. Ko, R. DeLine & G. Venolia. Information Needs in Collocated Software Development Teams, *ICSE*, 2007. 344-353.

[13] Ko AJ & Myers BA. Debugging, Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior, *ICSE*, 2008, 301-310, 2008.

[14] L. A. Miller. Natural Language Programming: Styles, Strategies, and Contrasts, *IBM Systems Journal*. 1981. **20**(2). 184-215.

[15] Miller P, Pane J, Meter G & Vorthmann S. Evolution of novice programming environments: The structure editors of Carnegie Mellon University. *Interactive Learning Environments*. 1994 **4**(2) 140-158.

[16] Murphy G, Lai, Walker R. & Robillard M. Separating Features in Source Code: An Exploratory Study, *ICSE*, 2001.

[17] Myers BA. Incense: A System for Displaying Data Structures, *Computer Graphics*, SIGGRAPH'83, 1983, 115-125.

[18] Myers BA. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy, *CHI 1986*, 59-66.

[19] Myers BA, Burnett MM, Wiedenbeck S & Ko AJ. End User Software Engineering. CHI'2007 *Extended Abstracts*, 2125-2128.

[20] Nardi BA. *A Small Matter of Programming: Perspectives on End User Computing*. The MIT Press. 1993.

[21] Pane JF & Myers BA. *Usability Issues in the Design of Novice Programming Systems*. Pittsburgh, PA, Carnegie Mellon University. CMU-CS-96-132. August, 1996.

[22] D.E. Perry, N.A. Staudenmayer & L.G. Votta. People, Organizations and Process Improvement, *IEEE Software*. 1994. 36-45.

[23] Scaffidi C, Shaw M & Myers B. Estimating the Numbers of End Users and End User Programmers, *VL/HCC'05*, 2005. 207-214.

[24] Shneiderman B. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, 1980.

[25] Shu NC. *Visual Programming*. New York, Van Nostrand Reinhold Company. 1988.

[26] Sillito J, Murphy GC & De Volder K. Questions programmers ask during software evolution tasks, *FSE*, 2006, 23 - 34.

[27] Soloway E. & Ehrlich K. Empirical Studies of Programming Knowledge, *TSE* **SE-10** 595-609, 1984.

[28] Stasko JT. Using Direct Manipulation to Build Algorithm Animations by Demonstration, CHI, 1991, 307-314.

[29] Tullio J, Dey AK, Chalecki J & Fogarty J. How IT works: a field study of non-technical users interacting with an intelligent system, *CHI*, 2007. 31-40.

[30] Weinberg GM. *The Psychology of Computer Programming*. New York, NY, von Nostrand Reinhold, 1971.