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 converging on problems of customization and end users psychology 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
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 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**


