View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Barriers to Successful End-User Programming

Andrew Ko
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
ajko@cs.cmu.edu, http://www.cs.cmu.edu/~ajko

In my research and my personal life, I have come to know numerous people that our research community might call end-user programmers. Some of them are scientists, some are artists, others are educators and other types of professionals. One thing that all of these people have in common is that their goals are entirely unrelated to producing code. In some cases, programming may be a necessary part of accomplishing their goals, such as a physicist writing a simulation in C or an interaction designer creating an interactive prototype. In other cases, programming may simply be the more efficient alternative to manually solving a problem: one might find duplicate entries in an address book by visual search or by writing a short Perl script.

In either case, the fact that end-user programmers are motivated by their domain and not by the merits of producing high-quality, dependable code, means that most of the barriers that end users encounter in the process of writing a program are perceived as distractions. This is despite the fact that such barriers can represent fundamental problems in end-users' program's or their understanding of how to use a programming language effectively.

Much of my research has focused on understanding these barriers and how end users overcome them. When are they insurmountable and why? What happens when end users fail to overcome them? And how can tools help end-user programmers' improve their programs' dependability, while allowing them to remain focused on their goals, rather than their code?

Studies of Barriers

Some of my earlier investigations of these barriers involved observations of non-programmers using the Alice programming environment to create interactive 3D worlds (Ko and Myers 2005). Some of these observations were done in the field, in the context of teams of students, only one of which was programming, and other observations were performed in a lab, with an experimenter. There were several barriers that users encountered that seemed fundamental to programming and programming tools, and not just to Alice. For example, *premature commitment* was a major problem in numerous contexts: users were forced to make decisions before they had enough information to do so accurately. For example, they had to create an object before they could write code to manipulate it. Or, when a user was trying to diagnose their program's failure, they had to base their hypothesis of what caused the failure just on what they could see in the program's output, rather than on information about the program's execution. In many of these situations, users premature decisions led to errors.

These observations led to broader study, aimed at classifying major barriers (Ko, Myers and Aung 2004). I observed over thirty students learning to use Visual Basic.NET to create simple form-based applications and user interfaces. I attempted to document the barriers that students encountered by telling them that they could consult the teaching assistants with any problems they felt they could not overcome. When consulted, the teaching assistants recorded the problem that the student was stuck on and the strategies that the student had used to try to overcome it. After classifying all of the different barriers that students encountered, there were six major barriers that accounted for our data:

Design – Complex computational problems that users were not trained to solve, such as sorting and searching.

Selection – Finding code, usually part of an API, that produces a desired behavior, such as tracking time.

Use – Once some class, method, or data structure was found, learning how to properly use its programming interface, such as how to start and stop a timer.

Coordination – Learning rules about how entities can communicate, such as how to send data between forms.

Understanding – Forming hypotheses about the potential causes of a program's behavior.

Information – Gathering information to test hypotheses about the causes of a program's behavior.

These six barriers accounted for all of the situations we observed in our study, and we have continued to observe them in other languages and tools.

Dagstuhl Seminar Proceedings 07081
End-User Software Engineering
http://drops.dagstuhl.de/opus/volltexte/2007/1091

The Whyline

In addition to studying the barriers that end user programmers face, I have also attempted to lower them with tools. The Whyline (Ko and Myers 2005) is aimed at alleviating difficulties with the *understanding* and *information* barriers described above, specifically for the Alice programming environment. Essentially, it allows users to choose some aspect of the program's behavior, such as a change to the color of some object onscreen, and ask *why* and *why not* questions about it. The Whyline then gives answers in terms of a causal chain of events that caused or prevented the behavior to occur. In a user study it was highly effective, reducing debugging time by a factor of 8. The reasons for this improvement were simple. By allowing users to reason about the output of their program, it deferred the premature formation of hypotheses about the causes of the behavior until the Whyline provided information, helping lower the *understanding* barrier. By providing the information about the program's execution automatically, rather than having users gather it manually, it almost entirely eliminated the *information* barriers that we observed in our earlier study of Alice.

Future Directions

My studies of barriers in end-user programming revealed many important problems to address, and the Whyline demonstrates one example of addressing them. However, not only are there many other barriers that deserve attention, but the tools that we design to help with each of these are influenced by a number of factors for which we still have little knowledge.

For example, the generalizability of any end-user software engineering tool depends greatly on the similarity of the work contexts of the end users we intend to design for. The Whyline was designed for a single user; in a group context, where many people may be involved in diagnosing and fixing a bug, the tool suddenly has many shortcomings. Do end user programmers work in groups? If they do, how is the work divided? What information do they share?

Another issue that may vary across different work contexts is the set of languages and applications with which end users' programs must interact. We might be able to design tools for one language, but can we design general tools to support Excel scripters interacting with a proprietary internal company database? To what extent do such setups actually occur for end users?

Although end-user programming language design has received much attention in the past, there are still several important issues to understand. For example, to what extent must a language match the work that end-users do? How can we help end users bridge the expressive gap in the languages they use and the behaviors they want to express? Because end users often lack the training to create the abstractions necessary to bridge these gaps, this will continue to be an issue.

Another software engineering issue that end users may encounter are the long-term maintenance issues common to commercial software development. We frequently hear anecdotes about how a one-off excel spreadsheet meant to be temporary became the centerpiece of some accounting logic. How often do such organizational dependencies occur, and how important do such program's become? What can tools do to help the future owners of these programs learn about the program's history and design?

Finally, one challenge about end-user programming is that end-user programmers needs may vary so widely that we cannot design tools and languages general enough, yet specific in their aid to help everyone. Do we approach this problem by simplifying the creation of end-user programming environments and creating highly tailored languages on-demand, by helping end-users bridge expressive gaps in a smaller number of languages, or by some other means? What general research contributions can we make and what specifics do we have to leave to individuals and the market?

That we face so many complex issues is encouraging. Not only does this mean that we have lots of interesting work to do, but it also means that we are closer to addressing real concerns. Let us continue to tackle them with rigor and objectivity.

Acknowledgements

This work was funded in part by the National Science Foundation (NSF) under grant IIS-0329090, the EUSES consortium under NSF grant ITR CCR-0324770, and an NDSEG fellowship.

References

Ko, A. J. and Myers, B. A. (2005). A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages and Computing*, 16, 1-2, 41-84.

Ko, A. J. Myers, B. A., and Aung, H. (2004). Six Learning Barriers in End-User Programming Systems. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Rome, Italy, September 26-29, 199-206.

Ko, A. J. and Myers, B. A. (2004). Designing the Whyline: A Debugging Interface for Asking Questions About Program Failures. *ACM Conference on Human Factors in Computing Systems*, Vienna, Austria, April 24-29, 151-158.