End-User Programming Productivity Tools

Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Jeffrey Stylos
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213
ajko@cs.cmu.edu, bam@cs.cmu.edu, mcoblenz@andrew.cmu.edu, jsstylos@cs.cmu.edu
http://www.cs.cmu.edu/~marmalade

ABSTRACT
Our research focuses on developing interactive technologies for a broad range of end-user programming activities, including code construction, verification, debugging, and understanding. A common goal among all of these technologies is to identify core ideas that can be used across a variety of domains and programmer populations.

INTRODUCTION
Although end-user programmers' interests vary widely, spanning the web, animation, documents, databases, mail, and countless other types of information, all of these users use programming as a means to an end [10]. Therefore, to minimize the distractions from end users' primary goal, it is essential that end user programming tools are approachable, easy to learn, and immediately helpful [1].

We are designing several technologies that satisfy these criteria, including new interaction techniques for editing code, new languages that help end users identify mistakes, debugging tools that answer users' questions about their program's output, and workspaces that help them understand the answers. All of these technologies have been directly inspired by the empirical research of a variety of programmer populations and their difficulties [5, 6, 8, 11].

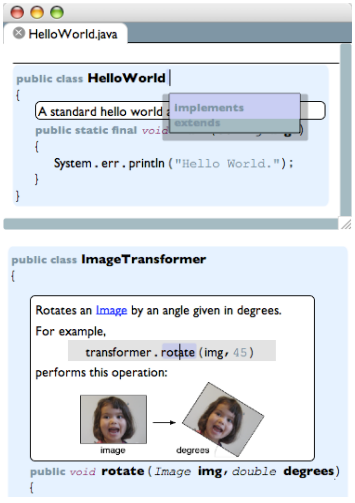

Figure 1. Barista [7], a Java editor that supports drag and drop, auto-complete menus, and text editing in a single editor, and embedded, in-context tools and visualizations.

CONSTRUCTING PROGRAMS
Syntax has long been a significant learning barrier in end-user programming systems, largely because of the difficulty of understanding and remembering the hidden and complex rules encoded in language grammars [5]. We have been working on a new class of code editors that try to help users construct code by choosing from different options rather than having to memorize the syntax. Barista [7], shown in Figure 1, is a Java editor that embodies this approach. It supports drag and drop interactions for creating and modifying code and syntactic and semantic auto-completion, as well as traditional text editing interaction techniques, all in a modeless editor. Barista also allows designers of end-user programming systems to embed tools and information in code, as illustrated by the method header on the bottom of Figure 1.

Although Barista is currently for Java, its underlying design and techniques could be an alternative to conventional text editors across the spectrum of programming languages.

DETECTING ERRORS
Some spreadsheet systems allow users to specify units (e.g. 5 lbs.) with their data in order to help detect unit errors in calculations. However, most data represented in spreadsheets is a measurement of a particular kind of object (e.g., 5 lbs of *apples*), and it is often inappropriate to perform calculations on data that represent different kinds of objects. Slate [2], shown in Figure 2, allows users to

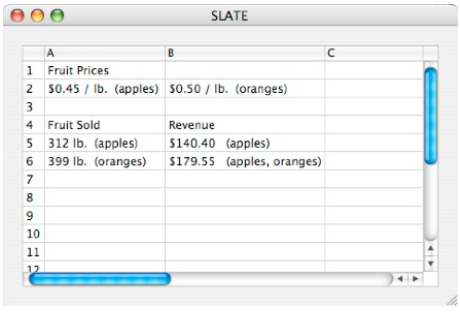

Figure 2. Slate [2], a spreadsheet language that allows users to give data *labels*, in order to help identify incorrect input and formulas. For example, the label "(apples, oranges)" at the bottom right of the spreadsheet suggests an error, since nothing can be apples and oranges simultaneously.

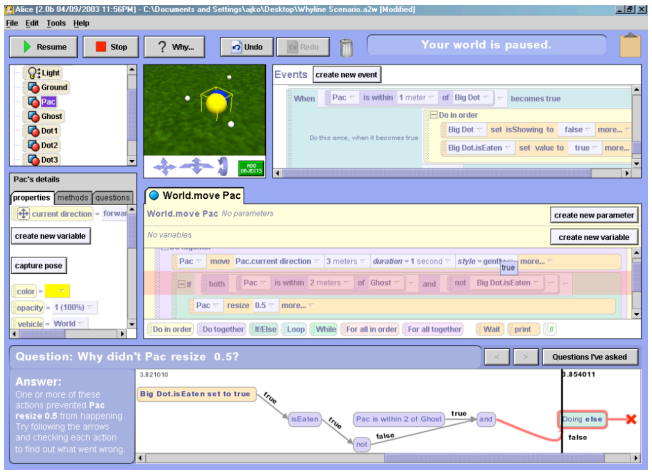

Figure 3. The Whyline [4] which allows users to ask "Why Did" and "Why Didn't" questions about their program's output, and get answers in terms of the events related to the behavior in question. In this situation, the user asked why Pac did not resize, and the answer shows the execution events that caused the "else" part of the conditional to be executed.

represent the object of measurement as a *label*. By intelligently propagating labels, Slate can help users identify incorrect input data and calculations. For example, in the spreadsheet shown in Figure 2, the result "$179.55 (apples, oranges)" tells the user that one of the formulas is likely to be incorrect, since nothing can be an apple *and* an orange at the same time.

Labels could be used in other end-user domains, such as animations or dynamic web pages that involve computation on heterogeneous and semi-structured data.

DEBUGGING PROGRAMS

One reason debugging is the most time-consuming part of programming is that end users must map their questions about a program's behavior onto debugging tools' limited support for analyzing code. We have been working on a new approach called *interrogative debugging,* which allows programmers to ask questions directly about their programs' output. Our prototype, the Whyline [4], allows programmers to ask "Why did" and "Why didn't" questions about their program's output in the Alice programming environment (www.alice.org). Programmers choose a question from an automatically generated menu, and the tool provides an answer, as seen in Figure 3, in terms of the runtime events that caused or prevented the desired output. In user studies of the Whyline, users with the Whyline spent an eighth as much time debugging the same bugs than users without the Whyline and made 40% more progress.

In generalizing the Whyline, we have begun to apply its ideas to traditional user interfaces. Our Crystal word processor [9], seen in Figure 4, allows users to ask questions such as "Why did this word change from 'teh' to 'the'?" and get answers in terms of the user interface components and state that were responsible for the word

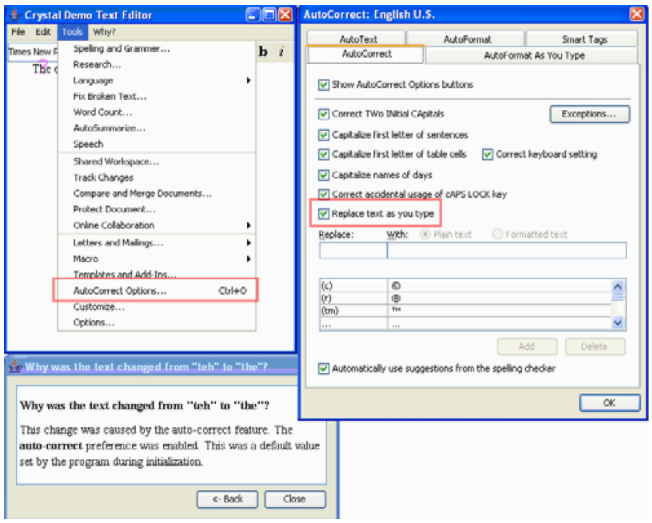

Figure 4. Crystal [9], a word processor that allows users to ask questions about the document and application state, and get answers in terms of the user interface components that are related to the behavior in question. In this situation, a user asked why a word changed from "teh" to "the", and the answer explains that the "Replace text as you type" checkbox is checked.

processor's behavior. A user study demonstrated that this helped users solve common problems about 30% faster than the same word processor without support for questions [9]. We are currently generalizing the Whyline to more complex and widely used languages, such as Java, in order to identify issues of scale and assess the range of questions that people ask about program behavior.

UNDERSTANDING PROGRAMS

Even though their programs tend to be small, end users still tend to have difficulty relating code to its corresponding behavior [5]. Furthermore, the interfaces that end users use to navigate and understand code, mainly windows and tabs, incur significant navigational overhead [6]. We are currently designing a new type of workspace that helps users both interactively and automatically collect fragments of code and other information that is relevant to their maintenance or debugging tasks. It will eliminate much of the navigational overhead, while helping users to quickly understand dependencies between different parts of their program.

LEARNING TERMINOLOGY

One common programming activity, even among end-user programmers [5] is learning to use a collection of external code in the form of libraries, toolkits, APIs, and frameworks. Some of the difficulty in this task comes from the fundamental *vocabulary problem* [3]: a particular programming concept can be described in multiple ways and no one word will best describe it for all programmers. Mica, shown in Figure 5, attempts to solve this problem by acting as a thesaurus: programmers supply a description of the desired functionality, using their own terminology, and

Figure 5. The Mica web application. Mica includes a keyword sidebar on the left, which is generated from Google Web API search results shown on the right. Search result pages containing code are marked with an icon.

Mica finds related classes and methods in the standard Java APIs in the form of keywords (method, class and interface names on the left in Figure 5) and regular web search results (on the right in Figure 5). Mica determines API keywords by analyzing the content of the Google search result pages and comparing these to a list of all class and method names for the standard Java API. The keywords are ranked based on the frequency with which they appear in the search result pages for the query and the overall frequency with which they appear on all pages indexed by Google. The list of keywords dynamically updates as Mica loads and processes all of the search result pages.

We plan to expand Mica's to aid other aspects of API use, such as understanding high-level API concepts, finding example code, and integrating examples into programs.

CONCLUSIONS

Our research covers a broad spectrum of programming activities, and we anticipate that our techniques will generalize to a variety of domains and programmer populations. We hope that our broad focus will both inspire new ideas for commercial programming tools and drive innovations in end user software engineering research.

ACKNOWLEDGMENTS

We thank our collaborators, including Htet Htet Aung, Christopher Scaffidi, and David Weitzman. This work was supported by the National Science Foundation, under NSF grant IIS-0329090, and as part of the EUSES consortium (End Users Shaping Effective Software) under NSF grant ITR CCR-0324770. The first author was supported by an NDSEG fellowship.

REFERENCES

1. Blackwell, A., First Steps in Programming: A Rationale for Attention Investment Models, *IEEE Symposia on Human-Centric Computing Languages and Environments*, (2002), 2-10.
2. Coblenz, M. J., Ko, A. J., and Myers, B. A., Using Objects of Measurement to Detect Spreadsheet Errors, *IEEE Symposium on Visual Languages and Human-Centric Computing*, (2005), 314-316.
3. Furnas, G. W., Gomez, T. K. L. L. M., and Dumais, S. T., "The Vocabulary Problem in Human-System Communication," in Communications of the ACM, 30, 1987, 964-971.
4. Ko, A. J. and Myers, B. A., Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior, *Human Factors in Computing Systems*, (2004), 151-158.
5. Ko, A. J., Myers, B. A., and Aung, H., Six Learning Barriers in End-User Programming Systems, *IEEE Symposium on Visual Languages and Human-Centric Computing*, (2004), 199-206.
6. Ko, A. J., Aung, H., and Myers, B. A., Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks, *International Conference on Software Engineering*, (2005), 126-135.
7. Ko, A. J. and Myers, B. A., Barista: An Implementation Framework for Enabling New Interaction Techniques and Visualizations in Code Editors, *ACM Conference on Human Factors in Computing*, (2005), to appear.
8. Ko, A. J. and Myers, B. A., A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems, *Journal of Visual Languages and Computing*, 16, 1-2, (2005), 41-84.
9. Myers, B. A., Weitzman, D. A., Ko, A. J., and Chau, D. H., Answering Why and Why Not Questions in User Interfaces, *ACM Conference on Human Factors in Computing Systems*, (2005), to appear.
10. Nardi, B. A., A Small Matter of Programming: Perspectives on End User Computing. Cambridge, MA: The MIT Press, 1993.
11. Panko, R., What We Know About Spreadsheet Errors, *Journal of End User Computing*, 2, (1998), 15-21.