Preserving Non-Programmers' Motivation with Error-Prevention and Debugging Support Tools

Andrew Jensen Ko
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
ajko@cmu.edu

Abstract

A significant challenge in teaching programming to disadvantaged populations is preserving learners' motivation and confidence. Because programming requires such a diverse set of skills and knowledge, the first steps in learning to program can be highly error-prone, and can quickly exhaust whatever attention learners are willing to give to a programming task. Our approach to preserving learners' motivation is to design highly integrated support tools to prevent the errors they would otherwise make. In this paper, the results of a recent study on programming errors are summarized, and many novel error-preventing tools are proposed.

1. Introduction

In teaching programming, low motivation is a significant challenge to overcome. Programming is an inherently complex activity, requiring knowledge and skills in architecture and algorithm design as well as scientific debugging skills such as data collection and hypothesis testing [1]. Because programming requires such a diverse skill set, learners' initial attempts at programming can be highly error-prone—and errors can cause a great deal of confusion and uncertainty, quickly consuming their confidence and motivation. This challenge is even greater for disadvantaged populations, who tend to have less confidence to begin with.

Our approach to help overcome this motivational barrier focuses on designing highly integrated tools to prevent programming errors. In doing so, we hope to prevent confusion and uncertainty and preserve learners' motivation, curiosity, and confidence.

2. The Programming Domain

We are currently focused on the domain of event-based programming, which is common in many systems such as Macromedia Director and Microsoft Visual Basic. We are studying Alice [2], a 3D programming system (Figure 1), which supports a concurrent, event-based programming language. Alice prevents syntax and type errors with a structured, drag-and-drop editing environment, preventing all syntax and type errors. In recent studies, teaching Alice instead of Java raised at-risk students' average grade from 1.3 to 2.8 on a 4.0 scale, bringing them to parity with non-at-risk students, and decreased at-risk student attrition from 90% to 10% [3].

Our current efforts are centered on the prevention of common, higher-level errors that we have observed in a series of observational studies.

3. Observations of Non-Programmers

Using the method of contextual inquiry, we have studied the activities of at-risk children at a local middle school and highly educated students (Carnegie Mellon students with varying programming experience) [4]. Most of the highly educated students lacked the knowledge and skills required to succeed, but had the motivation to complete their tasks. Nevertheless, the students still spent an average of 50% percent of their time debugging errors caused by premature design decisions and inadequate programming strategies. The at-risk children were excited by the domain content and spent half an hour at simple programming tasks each week. However, they were overwhelmed by errors caused by a lack of programming

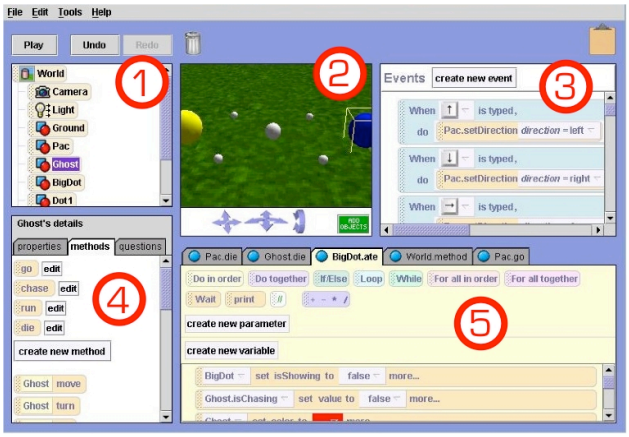

Figure 1. Alice, providing (1) an object list, (2) a 3D worldview, (3) an event list, (4) details about the selected object, and (5) methods being edited.

knowledge and quickly lost the motivation to complete even simple programming tasks. This was despite the fact that *all* syntax and type errors were prevented.

In our observations, we identified many key concepts and skills required to successfully create interactive worlds with Alice, as well as the *lack* of concepts and skills that tended to cause learners to commit errors. In general, there were three programming activities that required different types of knowledge and skills:

- *Design activities*, which required participants to have the foresight to understand the impact of design decisions on later implementation tasks.
- *Implementation activities*, which required knowledge of algorithms and code construction, modification, and reuse skills.
- *Debugging activities*, which required the ability to form hypotheses, gather data about the behavior of their programs, and make deductions from these observations to the potential causes of errors.

These observations resulted in a formal model of the causes of programming errors [4], which supports the design of error-preventing programming tools.

4. Opportunities to Prevent Errors

Using our observations and causal error model, we have identified common error situations in creating Alice programs that could be prevented with supportive, integrated environmental tools.

For example, learners tended to make design decisions about concurrency that were difficult to change and caused unforeseen interactions with other decisions. One approach to prevent these errors would be to embed design strategies in a storyboarding tool to help learners make concurrency decisions *before* implementation activities. These tools could have knowledge of common high-level design errors so that when learners make a problematic design decision, the system could explain the problem and suggest an alternative design. This would teach learners better design skills, while decreasing the likelihood of design errors and frustration.

Learners also tended to copy event handlers and animations but neglect to edit references to match the new context. This might be alleviated by visually highlighting copied code, indicating its *unreliability*, much like the testing and assertion systems in the Forms/3 end-user software engineering spreadsheet environment [5]. The learner could ask the system for an explanation, which would describe possible reasons for the code's unreliability. The tool could also interactively change references based on the new context and identify interactions, prompting the learner for verification. This would improve the correctness of learners' code while teaching them strategies for avoiding errors from duplicating code.

Because of learners' design and implementation errors, debugging was also a significant difficulty. Furthermore, though learners recognized when their programs failed, they had error-prone strategies and knowledge for determining which program statements had caused the program state that caused the failure. In this case, we could provide tools with embedded strategies and knowledge. For example, one tool might allow learners to tell the system when their program has failed, and in response, the system could present a timeline visualization of the concurrent threads of execution. The visualization could highlight time ranges with operations on shared data, and link these ranges to code. This could enable learners to find the incorrect areas of their code.

Another tool could allow learners to ask the system *why* and *why not* questions, and the system could provide a set of reasonable explanations derived from analyzing the program. For example, the learner could select a menu item, "why did my object disappear?" and the system could reply, "this method recently set the ball's visible flag to false." Such a tool would teach learners which questions to ask in a variety of debugging scenarios.

5. Conclusion

The tools described above are examples of highly integrated environmental support aimed at preventing common errors in event-based programming systems. In addition to teaching strategies and knowledge for the future, we expect such tools to directly support learners' current tasks and remain useful as learners gain expertise.

Future work will involve the design and implementation of these tools, and the empirical validation of their ability to prevent errors and support debugging activities. We believe that by preventing errors, such tools will reduce learners' confusion and uncertainty, helping to preserve the motivation necessary for learning programming skills.

6. References

[1] A. Blackwell, "First Steps in Programming: A Rationale for Attention Investment Models," at IEEE Symposia on Human-Centric Computing Languages and Environments, Arlington, VA, pp. 2-10, 2002.

[2] M. Conway, et al., "Alice: Lessons Learned from Building a 3D System For Novices," at Proceedings of CHI 2000, The Hague, The Netherlands, pp. 486-493, 2000.

[3] S. Cooper, et al., "Teaching Objects-First in Introductory Computer Science," at Proceedings of ACM SIGCSE 2003, Reno, NV, pp., 2003.

[4] A. J. Ko and B. A. Myers, "Development and Evaluation of a Model of Programming Errors," at Human Centric Computing, Auckland, New Zealand (to appear), 2003.

[5] K. J. Rothermel, et al., "WYSIWYT testing in the spreadsheet paradigm: an empirical evaluation," at Proceedings of the 22nd international conference on Software engineering, Limerick, Ireland, pp. 230 - 239, 2000.