Designing a Flexible and Supportive Direct-Manipulation Programming Environment

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

Abstract

An important part of helping learners detect, repair and avoid software errors is providing semantic support for learners while they manipulate their code. Unfortunately, usability aspects of both textual and direct-manipulation environments limit their ability to provide such support. Preliminary findings from exploratory studies are discussed, and several design requirements for a more flexible and supportive programming environment are identified.

1. Introduction

Software errors are a significant learning barrier for novice and end-user programmers [2]. As part of project Marmalade (http://www.cs.cmu.edu/~NatProg), the Whyline [3] has proven highly effective at helping learners *find* and *repair* their errors once they already exist. However, an equally important part of lowering barriers to programming is helping learners *avoid* errors by providing an *interactively flexible*, but *semantically supportive* authoring environment. Learners need the freedom to write code in whatever order is the most natural, while still getting clear, consistent, and immediate feedback about the effect of their actions on a program's semantics.

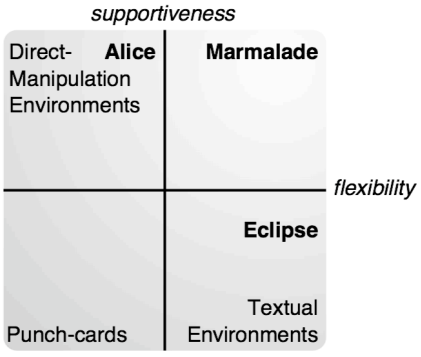

Figure 1. Programming environments in terms of interactive flexibility and semantic support.

Unfortunately, modern programming environments are rarely both flexible *and* supportive (as seen in Figure 1). Textual environments, at one extreme, allow learners to type code in whatever order is most natural. However, not only must learners memorize an awkward language syntax, but textual environments provide little support for helping learners remember it. Worse yet, textual environments fail to provide learners with immediate, consistent feedback on the effects of their actions on their program's semantics. Instead, learners are forced to decipher vague and misleading errors messages.

At the other extreme are direct-manipulation environments such as Alice [1]. These environments prevent all syntax and type errors, but require that programs always be in an executable state. This requirement has many implications: learners must create code in a strictly top-down manner, forcing awkward and premature decisions about their program's behavior; then, to modify code, they must translate the changes they want to make into a cumbersome series of state-preserving modifications.

The current goal of project Marmalade is to design an environment that is both flexible *and* supportive, while avoiding the downsides of these two extremes.

2. Approach

Our approach is to first study how people create, modify, and understand code. Specifically,

- At what *granularity* do learners manipulate code (character, token, statement, block, etc.)?
- What types of *local* modifications do learners perform (semantic, formatting, etc.)?
- What types of *global* manipulations do learners perform on code (changing method signatures, renaming variables, merging functionality, etc.)?
- How do learners *navigate* their code (by file, what calls what, what defines what, etc.)?

For each of these, it is important to compare what do learners currently do, what they want to do, as well as what their instructors would want them to do.

3. Preliminary Observations

We have observed several learners using Alice, VB.NET, Flash, and Eclipse, and have found many trends. For example, when learners worked with text, their code was almost always littered with compile-time errors. Because parsers are fundamentally limited in their ability to *fully* determine a syntactically ambiguous program's semantics, learners were often unable to get semantic support for even the *unambiguous* parts of their code. In Alice, which requires that programs *always* be executable, learners tended to rewrite code, since modifying it required first manually removing code that depended on it. This was more work, as well as considerably more error-prone.

In all environments, learners tended to leave uncoerced references in copied code, suggesting the need to explicitly mark copied code "suspect" until otherwise noted. Learners also tended to navigate semantic relationships rather than syntactic ones, such as method call to method definition and variable use to variable declaration. This suggests that these semantic relationships be quickly and explicitly navigable.

We have also found patterns in the *compile-time* and *runtime* correctness of learners' programs over time (portrayed in Figure 2). With text, delays in feedback due to compile-time errors caused debugging difficulties since learners falsely assumed that their most recent changes contained errors. This resulted in a pattern like that in Figure 2a. In Alice, there were no compile-time errors, but because of frequent rewriting, learners' programs had frequent drops in correctness (as in Figure 2b). We hope the pattern of correctness for our new environment to look like that in Figure 2c.

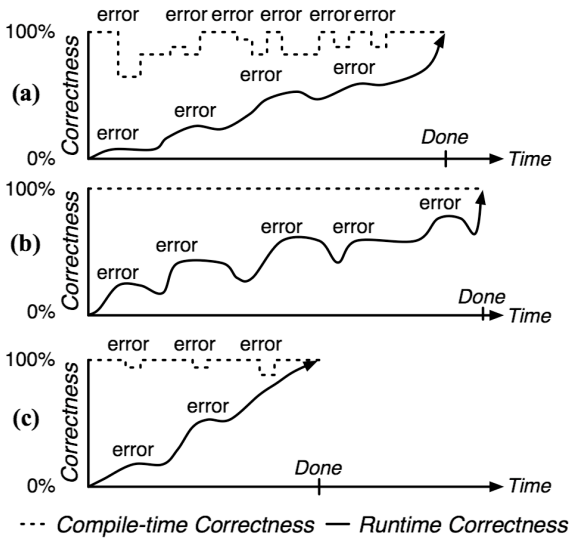

Figure 2. Mockups of patterns of correctness over time for (a) text and (b) direct-manipulation environments; (c) portrays one possible ideal.

4. Where is the Middle Ground?

The problems with textual and direct-manipulation environments suggest two overall requirements for a more helpful environment: (1) learners need semantic support for *all* of their code at edit-time, and a few compile-time errors should not affect this support; and (2) learners need the freedom to write code in *any* order and worry about compile-time errors later.

One possible solution would be an environment that (1) represents code *visually* as text, allowing for conventional text editing operations, (2) but represents code *internally* as higher-level semantic elements such as control and data flow graphs, and definitions and uses of data. This approach would preserve the interactive benefits of text editing that we have observed, but also enable more predictable and consistent semantic support than even the most sophisticated of textual environments, such as Eclipse. This is also significantly different from syntax-directed editors, which only manipulate a textual program's abstract syntax tree.

By separating the visual representation of code from its semantic elements, this environment could visualize programs as hierarchically indented text, flow charts, stacks of "tiles," or other forms depending on the context of use. For example, learners could see a view of an expression's dataflow with a single key; learners could hover over a variable to highlight the variable's definition, declaration, and other uses; the environment could even search for static definitions of variables that do not seem to be used by later code, and annotate such definitions in-context. We are not aware of any previous systems with such semantic support.

5. Conclusions

We are continuing to analyze our observations for design requirements, as well as consider possible software architectures for a more flexible and supportive programming environment. Although our current efforts will likely focus on helping novice Java programmers, we expect the new environment to support any type of end-user or expert language.

6. References

[1] Dann, W., Cooper, S., and Pausch, R. (2003). Learning to Program With Alice: Prentice-Hall.

[2] Ko, A. J. Myers, B. A., and Aung, H. (2004). Six Learning Barriers in End-User Programming Systems. Submitted to VL/HCC 2004.

[3] Ko, A. J. and Myers, B. A. (2004). Designing the Whyline: A Debugging Interface for Asking Questions About Program Failures. CHI 2004, Vienna, Austria, April 24-29, 151-158.