# Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing

**Amy J. Ko, Htet Htet Aung, and Brad A. Myers**
Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213
ajko@cmu.edu, hha@zizawah.com, bam+@cs.cmu.edu

**ABSTRACT**
A detailed study of Java programmers' text editing found that the full flexibility of unstructured text was not utilized for the vast majority of programmers' character-level edits. Rather, programmers used a small set of editing patterns to achieve their modifications, which accounted for all of the edits observed in the study. About two-thirds of the edits were of name and list structures and most edits preserved structure except for temporary omissions of delimiters. These findings inform the design of a new class of more flexible structured program editors that may avoid well-known usability problems of traditional structured editors, while providing more sophisticated support such as more universal code completion and smarter copy and paste.

**Author Keywords**
Structured editors, interaction techniques.

**ACM Classification**
D.2.6. Programming environments; H.5.2 User Interfaces.

**INTRODUCTION**
In the past decade, programmers have adopted an increasing number of tools that help create, modify, and navigate code in a more structured manner. For example, the Eclipse IDE offers auto-indenting and formatting, code completion tools to help write method calls, refactoring tools that help rename program elements and change method signatures, and searching tools that help find uses of variables, methods, and classes.

Unfortunately, most of these features fail in the presence of syntax ambiguities. Since the late 1970's, structured editors such as the Cornell Program Synthesizer [4], MacGNOME [3], and Alice [1] environments have escaped this limitation by avoiding parsing altogether, instead allowing the direct editing of the abstract syntax tree that represents a program. Not only does this persistent, structured representation of code prevent all syntax errors, but it also enables new ways of visualizing and operating on programs that are difficult or impossible with unstructured text.


*CHI 2005*, April 2–7, 2005, Portland, Oregon, USA.
ACM 1-59593-002-7/05/0004.

We are currently designing new tools that also require persistent structure, including universal code completion, more immediate feedback and help with type errors, and smarter copy and paste tools that maintain relationships between original and copied code [2]. Unfortunately, while persistent structure can help enable these innovations, it has traditionally introduced a significant usability problem [3]: because programmers are forced to use a top-down interaction technique for every edit, many modifications to code are more cumbersome than they would be with unstructured text. For example, changing a `while` loop to an `if` conditional in a structured editor requires creating the conditional, moving the body of the while loop to the conditional, and deleting the loop. The same modification in an unstructured text editor would have only required changing the keyword `while` to `if`.

One approach to designing a more flexible structured editor is to try to support the same editing strategies that programmers use for unstructured text, but with interaction techniques that preserve structure and are just as fast or faster to use than those offered by text editors. To test the feasibility of this approach, we performed a study of precisely how expert Java programmers utilize the flexibility of unstructured text, focusing on three questions:

1. What types of low-level changes do programmers make to code?
2. While making these changes, what intermediate states do programmers pass through and in what order?
3. What user interfaces do programmers use to perform these changes and why?

This paper reports on the identification of a small set of editing patterns that account for all of the edits observed in our study. These patterns suggest that the full flexibility of unstructured text is not required for most of the modifications that programmers make to code.

**METHOD**
The study was performed in the lab. Programmers were asked to complete five maintenance tasks on a 503-line Java painting program in a 70-minute period using the Eclipse 2.0 IDE. Three were debugging tasks, requiring single-line changes, and two were enhancements, requiring the creation and modification of classes, algorithms, and variables. We

| Edit | Structure UI | | State |
|---|---|---|---|
| | | | constructor.setColor(new Color(rSlider.getValue(), gSlider.getValue(), bSlider.getValue()));
| 1 | name | overwrite | constructor.~~setColor~~t(new Color(rSlider.getValue(), gSlider.getValue(), bSlider.getValue()));
| 2 | name | insert | constructor.t**hickness**(new Color(rSlider.getValue(), gSlider.getValue(), bSlider.getValue()));
| 3 | new | backspace | constructor.thickness(~~new Color(~~rSlider.getValue(), gSlider.getValue(), bSlider.getValue()));
| 4 | name | insert | constructor.thickness(**thick**Slider.getValue(), gSlider.getValue(), bSlider.getValue()));
| 5 | list | backspace | constructor.thickness(thickSlider.getValue()~~, gSlider.getValue(), bSlider.getValue()~~);
| | | | constructor.thickness(thickSlider.getValue());

**Figure 1. Five edits from a programmer's transcript. The 1st and 2nd changed the method name from `setColor` to `thickness`; the 3rd backspaced across structural boundaries, deleting the `new` operator (except its right parenthesis); the 4th changed a dot operator's left operand from `rSlider` to `thickSlider` and the 5th deleted the `new` operator's last two parameters and its dangling parenthesis.**

assumed that our choice of tasks would not influence programmers' character-level editing strategies.

We recruited 10 programmers with above-average self-rated Java expertise (mean 3.8, SD = 1.3) on a 1 to 5 scale. Programmers had a mean age of 21.6 years (SD = 2.6). We recorded a total of 700 minutes of screen-captured video of programmers' work, with 20% of the time spent editing; the majority of the remaining time was spent navigating and testing code. Programmers tried several solutions for each task, leading to significant diversity in edits.

We defined an *edit* as the result of inserting characters at the text caret, deleting or overwriting the text selection, backspacing over characters, undoing, pasting, or using code completion. The edits in each programmer's video were recorded in transcripts like the one in Figure 1, resulting in a set of 2770 edits. Although the number of programmers used to generate this data set was small, we believe that the data will generalize to any programmer with significant experience with the Java language syntax.

## RESULTS AND IMPLICATIONS

In this section, we identify patterns in programmers' editing strategies, recognizing that they may not generalize to languages with syntax that is widely different from Java's.

### Names

*Names* appear in declarations and references. Name edits were 43% of our data, and were one of seven types:

- *Creating a name*, by typing it from left to right (59%) or using code-completion (5%).
- *Replacing part of the internal structure of a name* (13%), as in replacing `colorPanel` with `strokePanel`.
- *Correcting typos*, by using the backspace key (10%).
- *Replacing a name*, by backspacing or selecting and deleting the entire old name first (4%).
- *Removing a name*, by backspacing, overwriting, or selecting and deleting (4%).
- *Splitting a name* (3%), as in splitting `padd` into `p.add`.
- *Renaming* (2%) a variable, method, or class and its uses.

About 25% of name edits resulted in *semantically invalid* names, because the edit was an intermediate change (49%), the name was undeclared (20%), or because of a typo (26%) or misspelling (5%).

These editing patterns show that programmers modify names in a variety of ways and create and modify names that are undeclared; such edits are prohibited in many modern structured editors, such as Alice [1]. These patterns also show that names contain more internal structure than current editors reason about; new structured editors may be able to take advantage of this structure in novel ways.

### Lists

In Java, *list* structures appear between list delimiters, such as the `{}`'s surrounding lists of statements and the `()`'s surrounding lists of parameters. List elements are delimited by single characters, such as the `;`'s between statements and the `,`'s between parameters. List edits accounted for 23% of our data, and were one of six types:

- *Creating a new list*, by typing the left list delimiter (32%); Eclipse automatically completed right delimiters.
- *Inserting a list element*, by typing the element delimiter first and then the element (9%), pasting list elements into a list (9%), or in the case of statements, inserting a blank line before typing the statement and the ';' (35%).
- *Removing a list element and its delimiter* (8%), by either backspacing, or selecting and deleting.
- *Moving the right list delimiter* (3%), by backspacing over it and then inserting it elsewhere, in order to include or exclude nearby elements from a surrounding list.
- *Removing an entire list and its elements* (2%), by either backspacing or selecting and deleting.
- *"Flattening" a list inside of a list* (2%), as shown in Figure 1, by removing the left and right list delimiters and any unwanted elements. The remaining elements from the "flattened" list were always left inside a surrounding list.

These editing patterns suggest that the arbitrary modification of lists may not be required. For example, support for moving or deleting the left list delimiter may not be required; instead, structured editors could offer interaction techniques to replace a list with a subset of its elements. Support for deleting the right list delimiter may also not be required; instead, structured editors could offer interaction techniques for moving the right delimiter to include or exclude code.

### Method Calls and Instantiations

Method calls have the form `name(parameters)` and instantiations have the form `new name(parameters)`. Because both have a name and parameter list, most edits to

these structures used the name and list edits described earlier. When created, they were always typed from left to right in their entirety. Several editing patterns applied directly to these structures, accounting for 1% of our data:

- *Creating a method call or instantiation* using code completion on an existing partial name (74%).
- *Replacing a method call or instantiation with one of its parameters* (17%), as in replacing `v.m(a, b)` with `b`, by backspacing, or selecting and deleting the text left and right of the desired parameter.
- *Removing an entire method call or instantiation* (9%) by selecting and deleting.

Despite the fact that method calls and instantiations have different semantics and that both are specialized Java structures, they were still edited as a simple name and list. This suggests that programmers' editing strategies had more to do with their structure than their semantics.

### Infix Expressions

In Java, infix expressions have the form *`operand operator operand`* and include arithmetic, Boolean, assignment, and dot operators. Infix edits accounted for 15% of our data, and were one of seven types:

- *Creating an infix operator* with only the left operand present, as in `obj.` (64%); with both operands present, as in typing a '`.`' after the `obj` in `objmethod` (5%); or by pasting a complete infix expression (1%). They were never created with only the right operand or neither operand present.
- *Replacing the infix expression with its left* (8%) *or right* (1%) *operand*, as in replacing `var.method` with `var`, by backspacing or selecting and deleting.
- *Re-activating code completion* (8%), by removing and re-inserting the dot operator.
- *Removing an entire infix expression* (5%), by backspacing, or selecting and deleting.
- *"Wrapping" a left or right operand* (4%), by inserting a parenthesis before and after the operand.
- *"Unwrapping" a left or right operand* (2%), by backspacing over the parentheses around the operand.
- *Changing the infix operator* (2%) to a semantically comparable alternative, such as changing `+` to `-`.

These patterns show that infix expressions were created in many different orders, but never operator first, which is exactly what traditional structured editors require. Also, when infix operators were inserted between operands that were not yet wrapped inside of parentheses, the structure was temporarily ambiguous. This ambiguity may require structured editors to allow "dangling structures" that are complete and valid, but not yet owned by a structure.

### Prefix Expressions

In Java, prefix expressions have the form *`operator operand`* and include operators such as the `!`, `-`, `++`, `--` and typecast operators. Prefix edits accounted for 1% of our data, and were one of five types:

- *Applying a prefix operator to an expression* (81%), by typing the operator name and then "wrapping" the operand in parentheses (if necessary), or vice versa.
- *"Unwrapping" a prefix operand* (11%), by removing the operator and then removing the parentheses around the expression (if necessary), or vice versa.
- *Removing an entire prefix expression* (7%), by backspacing, or selecting and deleting.
- *Changing a prefix operator* (1%) to semantically comparable alternative, such as changing the type of a type cast or a `++` to a `--`.

These editing patterns suggest that if structured editors offered interaction techniques for "wrapping" and "unwrapping" prefix operators around an expression, arbitrary parenthesis placement may not be required.

### Keyword Structures

Java's "keyword" structures, such as declarations and `if` and `while` constructs, involving one or more keywords, such as `class` or `extends`, as well as one or more names and lists. When created, they were always typed from left to right in their entirety. Several editing patterns applied directly to these structures, accounting for 6% of our data:

- *Typing a required keyword* (71%), in the process of typing a complete keyword structure from left to right.
- *Typing an optional keyword* (16%), such as `public` or `extends`, into an existing or new structure.
- *Creating optional structure* (7%), such as an initialization statement in a variable declaration.
- *Creating a declaration* with refactoring tools (4%).
- *Removing an entire keyword structure* (2%) by selecting its left and right extents and backspacing.

Programmers never modified required keywords, such as changing an `if` to a `while` loop (as in the introduction). Instead, programmers always deleted the existing structure and created the new one, typically because there was little in common between the existing code and the desired code.

Because keyword structures were always created, removed, and selected in their entirety, structured editors may not need to support their arbitrary modification or selection. The flexibility to type optional keywords may also not be required; instead keywords could be chosen from lists that can be "typed-through," easing their modification.

### Literals

Literals include strings (`"str"`), characters (`'c'`), integers (`12345`), floating-point numbers (`123.45`), and other constant-valued terms. Literal edits were 8% of our data, and were one of three types:

- *Creating a complete literal* (60%) by typing left to right, or as part of name edits which became infix expressions (as when `123.45myVar` becomes `123.45 + myVar`).
- *Modifying a literal* (27%), preserving its structure. For example, programmers never wrote part of a string as in `"an unfinished string`.
- *Replacing a literal with an expression* (13%) by first removing the entire literal.

### Comments

Java supports `//` comments, which exclude a complete line, and `/**/` comments, which exclude an arbitrary sequence of characters. Comment edits accounted for 3% of edits in our data, and were one of three types:

- *Temporarily commenting out statements* (60%) by typing a `//` before a statement, as in "`//repaint;`" and often creating an alternative statement above or below.
- *Creating annotations* (37%), which often referenced named program elements such as variables and classes, using either type of comment.
- *Temporarily replacing an expression* (3%), often in unstructured ways, but with structured intent, as in "`(a + 2);//b);`". These unstructured edits were likely due to the hassle of typing `/**/` comments within a line.

These edits suggest that arbitrary commenting of text may not be required. Instead, structured editors could offer interaction techniques that make it easy to comment out arbitrary structures. Also, because annotations often referenced names, the environment could give structure to comments as well; for example, clicking a name in a comment could navigate to the name's declaration.

### Undo

Programmers frequently made typos and more substantial editing mistakes that they later fixed. We identified two strategies for these repairs in our data:

- *Repairing typos* using the backspace key. This rarely occurred immediately after a typo, but instead after several other correct characters had been typed.
- *Repairing more substantial mistakes* by holding the backspace key, or selecting and deleting. Because these repairs frequently occurred after other correct edits were completed, programmers rarely used Eclipse's undo since it would have required undoing these correct edits.

Because programmers made typos and mistakes in every type of structure, structured editors need to support a deletion mechanism for every creation mechanism. For example, if an editor were to create a dot structure when a programmer types '.', there should also be a way to delete the structure using the keyboard. However, our evidence suggests that backspace was a versatile deletion mechanism because it only depends on the text caret position, and not on the editing history; thus, such deletion mechanisms should not depend on the editing history either.

### Text Selection

There were two circumstances where text was selected across structural boundaries:

- *Selecting for efficiency*. Two or more structural edits, as in the name change and list delimiter removal in Figure 1—were performed with one selection.
- *Multiple selection of list elements across list boundaries*, for example, selecting two statements before an `if` and two inside of it. The intruding text between the elements (in our example, the `if` header) was always deleted later.

These selections suggest that if structured editors offered multiple selection mechanisms that obeyed structural boundaries, many of the selections that programmers do in unstructured text would actually be easier to perform.

## CONCLUSIONS

Unstructured text provides programmers with considerable freedom, but in our study, little of this freedom was utilized to modify code. Rather, programmers used the small set of editing patterns identified in our study to achieve their modifications. When programmers did make unstructured edits, they were typos, which were quickly repaired, or temporary omission of delimiters. Thus, one way for structured editors to be as flexible as text editors may be to support the editing patterns that are enumerated in this paper with interaction techniques that are just as fast or faster to use than the techniques offered by text editors. We intend to design, implement and improve on such techniques through extensive user testing, while adding new features that are only possible with the persistent structure offered by structured editors.

## ACKNOWLEDGMENTS

We thank Scott Hudson, James Fogarty, Elsa Golden, Karen Tang, and Santosh Mathan for their help with the experiment design. This research was funded by the National Science Foundation under NSF grant IIS-0329090 and as part of the EUSES consortium under NSF grant ITR CCR-0324770.

## REFERENCES

[1] Kelleher, C., Cosgrove, D., Culyba, D., Forlines, C., Pratt, J., and Pausch, R., Alice2: Programming without Syntax Errors, *User Interface Software and Technology*, Paris, France, 2002.

[2] 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*, St. Louis,MI, to appear, 2005.

[3] Miller, P., Pane, J., Meter, G., and Vorthmann, S., Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University, *Interactive Learning Environments*, 4, 2, 140-158, 1994.

[4] Teitelbaum, T. and Reps, T., The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, *Communications of the ACM*, 24, 9, 563-573, 1981.