Deuce: A Lightweight User Interface for Structured Editing

Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh
University of Chicago
{brianhempel,justinlubin,gracelu,rchugh}@uchicago.edu

ABSTRACT

We present a structure-aware code editor, called Deuce, that is equipped with direct manipulation capabilities for invoking automated program transformations. Compared to traditional refactoring environments, Deuce employs a direct manipulation interface that is tightly integrated within a text-based editing workflow. In particular, Deuce draws (i) clickable widgets atop the source code that allow the user to *structurally select* the unstructured text for subexpressions and other relevant features, and (ii) a lightweight, interactive menu of potential transformations based on the current selections. We implement and evaluate our design with mostly standard transformations in the context of a small functional programming language. A controlled user study with 21 participants demonstrates that structural selection is preferred to a more traditional text-selection interface and may be faster overall once users gain experience with the tool. These results accord with Deuce's aim to provide human-friendly structural interactions on top of familiar text-based editing.

CCS CONCEPTS

• **Software and its engineering** → Integrated and visual development environments; • **Human-centered computing** → Human computer interaction (HCI);

KEYWORDS

Structured Editing, Refactoring, Direct Manipulation

ACM Reference Format:
Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: A Lightweight User Interface for Structured Editing. In *ICSE '18: 40th International Conference on Software Engineering , May 27–June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180165

1 INTRODUCTION

Plain text continues to dominate as the universal format for programs in most languages. Although the simplicity and generality of text are extremely useful, the benefits come at some costs. For novice programmers, the unrestricted nature of text leaves room for syntax errors that make learning how to program more difficult [Altadmri et al. 2016]. For expert programmers, many editing tasks—perhaps even the vast majority [Ko et al. 2005]—fall within specific patterns that could be performed more easily and safely by automated tools. Broadly speaking, two lines of work have, respectively, sought to address these limitations.

Structured Editing. Structured editors—such as the Cornell Program Synthesizer [Teitelbaum and Reps 1981], Scratch [Maloney et al. 2010; Resnick et al. 2009], and TouchDevelop [Tillmann et al. 2012]—reduce the amount of unstructured text used to represent programs, relying on blocks and other visual elements to demarcate structural components of a program (*e.g.* a conditional with two branches, and a function with an argument and a body). Operations that create and manipulate structural components avoid classes of errors that may otherwise arise in plain text, and text-editing is limited to within well-formed structures. Structured editing has not yet, however, become popular among expert programmers, in part due to their cumbersome interfaces compared to plain text editors [Monig et al. 2015], as well as their restrictions that even transitory, evolving programs always be well-formed.

Text Selection-Based Refactoring. An alternative approach in integrated development environments (IDEs), such as Eclipse, is to augment unrestricted plain text with support for a variety of refactorings [Fowler 1999; Griswold 1991; Roberts et al. 1997]. In such systems, the user text-selects something of interest in the program—an expression, statement, type, or class—and then selects a transformation either from a menu at the top of the IDE or in a right-click pop-up menu. This approach provides experts both the full flexibility of text as well as mechanisms to perform common tasks more efficiently and with fewer errors than with manual, low-level text-edits. Although useful, this workflow suffers limitations:

(1) The text-selection mechanism is error-prone when the item to select is long, spanning a non-rectangular region or requiring scrolling [Murphy-Hill and Black 2008].
(2) All transformations must require a single "primary" selection argument, and any additional arguments are relegated to a separate Configuration Wizard window.
(3) The list of tools is typically very long—even in the right-click menu where tools that are not applicable to the primary selection are filtered out—making it hard to *identify*, *invoke*, and *configure* a desired refactoring [Mealy et al. 2007; Murphy-Hill et al. 2009; Vakilian et al. 2012].
(4) Even when a transformation has no configuration options or when the defaults are acceptable—as is often the case [Murphy-Hill et al. 2009]—the user must go through a separate Configuration Wizard to make the change. The user must, furthermore, navigate to another pane within the Configuration Wizard to preview the changes before confirming them.

Our Approach. Our goal is to enable a workflow that enjoys the benefits of both approaches. Specifically, programs ought to be

ICSE '18, May 27–June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180165

represented in plain text for familiar and flexible editing by expert programmers, and the editing environment ought to provide automated support for a variety of code transformations without deviating from the text-editing workflow.

In this paper, we present a structure-aware editor, called Deuce, that achieves these goals by augmenting a text editor with (i) clickable widgets directly atop the program text that allow the user to *structurally select* the unstructured text for subexpressions and other relevant features of the program structure, and (ii) a *context-sensitive tool menu with previews* based on the current selections.

Structural Code Selection. Rather than relying on keyboard-based text-edits for selection, our editor draws direct manipulation widgets to *structurally select* items in the code with a single mouse-click. In particular, holding down the Shift key transitions the editor into structural selection mode. In this mode, the editor draws a box (which resembles a text-selection highlight) around the code item below the current mouse position. Clicking the box selects the entire text for that code item, eliminating any possibility for error and reducing the time needed to select long, non-rectangular sequences of lines. Furthermore, this interface naturally allows multiple selection, even when items are far apart in the code. Structural text selection helps address concerns (1) and (2) above.

Context-Sensitive Menu with Previews. Because structural selection naturally supports multiple selection, we address concern (3) by showing only tools for which *all* necessary arguments have been selected, reducing the number of tools shown to the user compared to a typical right-click menu. Hovering over a result description previews the changes, and clicking a result chooses it. For tools with few configuration options, we believe the preview menu provides a lightweight way to consider multiple options while staying within the normal editing workflow, helping to address concern (4).

The resulting workflow in Deuce is largely text-driven, but augmented with automated support for code transformations (*e.g.* to introduce local variables, rearrange definitions, and introduce function abstractions) that are tedious and error-prone (*e.g.* because of typos, name collisions, and mismatched delimiters), allowing the user to spend keystrokes on more creative and difficult tasks that are harder to automate. The name Deuce reflects this streamlined combination of text- and mouse-based editing.

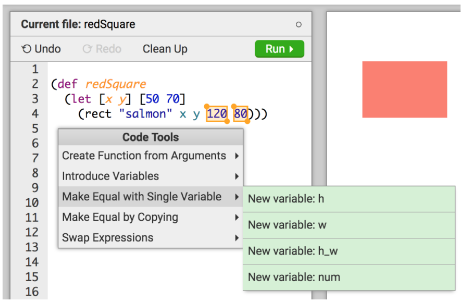
Contributions. This paper makes the following contributions:

- We present the design of Deuce, a code editor equipped with *structural code selection*, a lightweight direct manipulation mechanism that helps to identify and invoke program transformations while retaining the freedom and familiarity of traditional text-based editing. Our design can be instantiated for different programming languages and with different sets of program transformations. (§3.1)
- We implement Deuce within Sketch-n-Sketch, a programming environment for creating Scalable Vector Graphics (SVG) images. Most of the functional program transformations in our implementation are common to existing refactoring tools, but two transformations—*Move Definitions* and *Make Equal*—are, to the best of our knowledge, novel. (§3.2)
- To evaluate the utility of our user interface, we performed a controlled user study with 21 participants. The results show that, compared to a more traditional text selection-based refactoring interface, structural code selection is preferred and may be faster for invoking transformations, particularly as users gain experience with the tool. (§4)

Our implementation, videos of examples, and user study materials are available at http://ravichugh.github.io/sketch-n-sketch/. In the next section, we introduce Deuce with a few short examples.

2 OVERVIEW EXAMPLES

Example 1. Despite the intention of the following program, the `redSquare` definition uses different values for the width and height of the rectangle (the fourth and fifth arguments, respectively, to the `rect` function). The user chooses Deuce code tools—rather than text-edits—to correct this mistake.

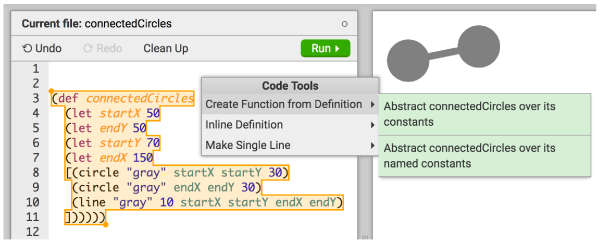


The user presses the Shift key to enter structured editing mode, and then hovers over and clicks the two constants `120` and `80` to select them; the selected code items are colored orange in the screenshot above. Based on these selections, Deuce shows a pop-up Code Tools menu with several potential transformations. The *Make Equal by Copying* tool would replace one of the constants with the other, thus generating a square. However, such a program would require two constants to be changed whenever a different size is desired. Instead, the user wishes to invoke *Make Equal with Single Variable* to introduce a new variable that will be used for both arguments. Hovering over this menu item displays a second-level menu (shown above) with tool-specific options, in this case, the names of four suggested new variable names.

The user hovers over the second option, which shows a preview of the transformed code (shown on the right). The user clicks to choose the second option. Notice that the number `80` (rather than `120`) was chosen to be the value of the new variable `w`. Whereas the tool provided configuration options for the variable name, it did not provide options for which value to use; this choice was made by the implementor of the *Make Equal* code tool, not by the Deuce user interface.

```
(def redSquare
  (let [x y] [100 70]
  (let w 80
    (rect "salmon" x y w w))))
```

Example 2. Consider the following program that draws two circles connected by a line. All design parameters and shapes have been organized within a single top-level `connectedCircles` definition. To make the design more reusable, the user wants `connectedCircles` to be a function that is abstracted over the positions of the two circles. The user hovers over and clicks the `def` keyword, and selects the *Create Function from Definition* tool (shown in the screenshot).



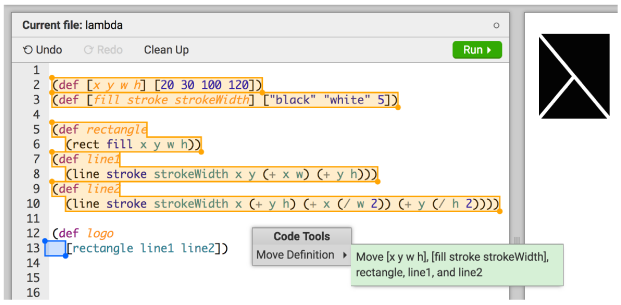
In response, *Create Function* rewrites the definition to be a function (shown on the right), and previous uses of `connectedCircles` are rewritten to appropriate function calls (not shown).

```
(def connectedCircles (\(startX endY startY endX)
  [(circle "gray" startX startY 30)
   (circle "gray" endX endY 30)
   (line "gray" 10 startX startY endX endY)
  ]))
```

Code Tools
Reorder Arguments

The order of arguments to the function match the order of definitions in the previous program, but that order was unintuitive—the coordinates of the `start` and end points were interleaved. To fix this, as shown above, the user selects the last two arguments and the *target position* (*i.e.* the space enclosed by a blue rectangular selection widget) between the first two, and selects the *Reorder Arguments* tool so that the order of arguments becomes `startX`, `startY`, endX, and endY (not shown). Calls to `connectedCircles` are, again, rewritten to match the new order (not shown).

Example 3. In the program below, the user would like to organize all design parameters and shapes within the single `logo` definition. The user hovers over and selects the five definitions on lines 2 through 9, as well as the space on line 13, and selects the *Move Definitions* tool to move the definitions inside `logo`. The transformation manipulates indentation and delimiters appropriately in the final code (not shown).



3 DEUCE: DESIGN AND IMPLEMENTATION

In this section, we explain the design of DEUCE in more detail. First, we define a core language of programs where various structural features can be selected. Then, we describe a user interface that displays active transformations based on the set of structural selections. Finally, we describe a set of general-purpose program transformations that are provided in our current implementation.

LITTLE. To make the discussion of our design concrete, we choose to work with a small functional language called LITTLE, defined in Figure 1. A LITTLE program is a sequence of top-level definitions, the last of which is called `main`. Notice that all (sub)expressions, (sub)patterns, definitions (both at the top-level and locally via `let`),

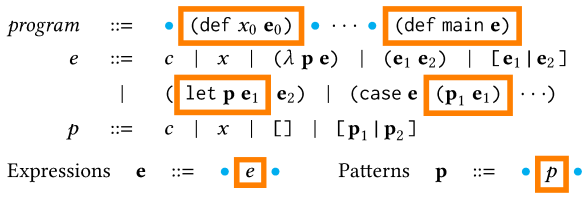


Figure 1: Syntax of LITTLE. The orange boxes and blue dots identify features for structural selection.

```
EditorState = { code: Program, selections: Set Selection }
ActiveState = Active | NotYetActive | Inactive
Options     = NoOptions | StringOption String
Result      = { description: String, code: Program }

CodeTool =
  { name : String
  , requirements : String
  , active : EditorState -> ActiveState
  , run : (EditorState, Options) -> List Result }
```

Figure 2: Code tool interface.

and branches of `case` expressions are surrounded in the abstract syntax by orange boxes; these denote *code items* that will be exposed for selection and deselection in the user interface. In addition, there are *target positions*, denoted by blue dots, before and after every definition, expression, and pattern in the program. Target positions are "abstract whitespace" between items in the abstract syntax tree, which will also be exposed for selection.

Code Tool Interface. Each code tool must implement the interface in Figure 2. A tool has access to the `EditorState`, which contains a `Program` and the `Set` of structural `Selections` within it. Based on the `EditorState`, the `active` predicate specifies whether the tool is `Active` (ready to run and produce `Result` options), `NotYetActive` (could be `Active` if given more valid selections), or `Inactive` (invalid based on the selections). For example, *Move Definitions* is `NotYetActive` if the user has selected one or more definitions but no target position. When invoked via `run`, a tool has access to the `EditorState` and configuration `Options`, namely, an optional `String`. This strategy supports the ubiquitous *Rename* tool. A more full-featured interface may allow a more general set of configuration parameters; the challenge would be to expose them using a lightweight user interface. In our implementation, all transformations besides *Rename* require `NoOptions`. Each `Result` is a new `Program` and a `description` of the changes.

This API between the user interface and code tool implementations is shallow, in the sense that a code tool implementation can do whatever it wants with the selection information. A framework for defining notions of transformation correctness would be a useful line of work, but is beyond the scope of this paper. Currently, code tools must be implemented inside the DEUCE implementation. Designing a domain-specific language for writing transformations would be useful, but is also beyond the scope of this paper.

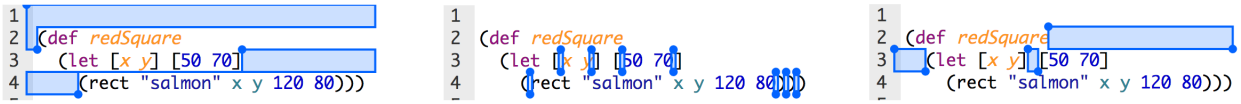


Figure 3: Example target positions.

Implementation in Sketch-n-Sketch. We have chosen to implement our design within Sketch-n-Sketch [Chugh et al. 2016; Hempel and Chugh 2016], an interactive programming system for generating SVG images. Whereas Sketch-n-Sketch provides capabilities for directly manipulating the *output* of a program, Deuce provides capabilities for directly manipulating the *code* itself.

Direct code manipulation is particularly useful for a system like Sketch-n-Sketch for a couple reasons. First, while the existing output-directed synthesis features in Sketch-n-Sketch attempt to generate program updates that are readable and which maintain stylistic choices in the existing code, the generated code often requires subsequent edits, *e.g.* to choose more meaningful names, to rearrange definitions, and to override choices made automatically by heuristics; Deuce aims to provide an intuitive and efficient interface for performing such tasks. Furthermore, by allowing users to interactively manipulate both code and output, we provide another step towards the goal of *direct manipulation programming systems* identified by Chugh et al. [2016]. These two capabilities—direct manipulation of code and output—are complementary.

Sketch-n-Sketch is written in Elm (http://elm-lang.org/), a language in which programs are compiled to JavaScript and run in the browser. The project uses the Ace text editor (https://ace.c9.io/) for manipulating Little programs. (The second reason for the name Deuce is that it extends Ace.) We extended Sketch-n-Sketch to implement Deuce; our changes constitute approximately 9,000 lines of Elm and JavaScript code. The new version (v0.6.2) is available at http://ravichugh.github.io/sketch-n-sketch/.

3.1 User Interface

The goals of our user interface are, first, to expose *structural code selection widgets*—corresponding to the code items and target positions in a Little program—and, second, to display an interactive menu of active transformations based on the set of selections.

So that the additional features provided by Deuce do not intrude on the text-editing workflow, we display structural selection widgets when hovering over the code box only when the user is holding down the Shift key. Hitting the Escape key at any time deselects all widgets and clears any menus, returning the editor to text-editing mode. This allows the user to quickly toggle between editing modes during sustained periods in either mode. When not using the Shift modifier key, the editor is a standard, monospace code editor with familiar, unrestricted access to general-purpose text-editing features.

3.1.1 Structural Code Selection. The primary innovation in our design is the ability to *structurally select* concrete source text corresponding to code items and target positions from the abstract syntax tree of a program.

Code Items. Our current implementation draws an invisible "bounding polygon" around the source text of each expression, which tightly wraps the expression even when stretched across multiple lines. These polygons serve as mouse hover regions for selection, with the polygons of larger expressions drawn behind the (smaller) polygons for the subexpressions such that all polygons for child expressions partially occlude those of their parents. Because complex expressions in Little are fully parenthesized, it is always unambiguous exactly where to start and end each polygon, and there are always character positions that can be used to select an arbitrary subexpression in the tree. Similarly, we create bounding polygons for all patterns and definitions.

When hovering over an invisible selection polygon, Deuce colors the polygon to indicate that it has become the focus. Its transparency and style is designed to resemble what might otherwise be expected for text selection (*cf.* the screenshots in §2). Clicking a polygon selects the code item, making it visible even after hovering away. Hovering the mouse back to the polygon and clicking it again deselects the code item.

Target Positions. The user interface also draws polygons for the whitespace between code items for selecting target positions. Figure 3 (left) shows how our implementation draws whitespace polygons slightly to the left of the beginning of a line, and until the end of a line even if there are no characters on that line. Figure 3 (center) shows whitespace polygons with non-zero width even when there are no whitespace characters between adjacent code items.

Another concern is that many target positions in the abstract syntax from Figure 1 describe the *same* space between code items. For example, the expression `[•50••70•]` on line 3 of Figure 3 contains both an after-`50` and before-`70` position. Because such target positions between adjacent items are redundant, our implementation draws only one whitespace polygon. (This polygon is not selected in any of the screenshots.)

A more interesting case is for the code items `(def •p••e•)` and `(let •p••e•···)`; there is both an after-**p** target and a before-**e** target. To allocate the whitespace between **p** and **e**, we take the following approach. The space up to the first newline, if any, is dedicated to after-**p**; the remaining is for before-**e**. If there is no newline, then we do not expose any selection widget for before-**e**. For comparison, notice how the whitespace from the end of line 2 to beginning of line 3 in Figure 3 (right) is split into two polygons, but the whitespace from the end of line 3 to the beginning of line 4 in the Figure 3 (left) is not. In other settings, it may be worthwhile to consider alternative approaches to the design decisions above.

3.1.2 Displaying Active Code Tools. Several program transformations may be `Active` based on the items and targets that are selected. We design and implement a lightweight user interface for identifying, invoking, and configuring `Active` transformations.

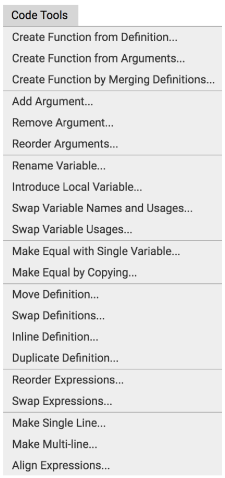
Pop-up Panel. When the user has entered structured editing mode (by pressing Shift) and selected at least one item, we automatically

display a menu near the selected items. The user has already pressed a key to enter this mode, so our design does not require a right-click to display the menu. The user can drag the pop-up panel if it is obstructing relevant code. We often manually re-positioned pop-up menus to make the screenshots in §2 fit better in the paper.

Hover Previews. Each tool in the menu has a list of `Results`, which appear in a second-level menu when hovering the tool `name`. Each second-level menu item displays the `description` of the change, and hovering over it previews the new code in the editor. Clicking the item confirms the choice and clears all Deuce selections. When there are few `Results` (*i.e.* configuration options), this preview menu provides a quick way to consider the options, rather than going through a separate Configuration Wizard. For tools that require multiple and non-trivial configurations, however, the editor could fall back on separate, tool-specific Configuration Wizards; our current implementation of Deuce does not support this.

3.2 Program Transformations

We have implemented a variety of program transformations, shown on the right. While we believe these transformations form a useful set of basic tools for common programming tasks, we do not argue that these constitute a necessary or sufficient set. One benefit of our design is that different sets of transformations—such as refactorings for class-based languages [Fowler 1999], refactorings for functional languages [Thompson and Li 2013], transformations that selectively change program behavior [Reichenbach et al. 2009], and task-specific transformations that do not have common, recognizable names [Steimann and von Pilgrim 2012]—can be incorporated and displayed to the user within our interface.



We limit our discussion below to transformations that are not implemented in existing refactoring tools. The Supplementary Appendices [Hempel et al. 2018] describe other transformations, but these details are not necessary to understand the rest of the paper.

Make Equal with Single Variable. When multiple constants and an optional target position are selected, the *Make Equal with Single Variable* transformation introduces a new variable, bound to one of the constants, and replaces all the constants with the new variable. This has the effect of changing the program to make each of these values equal. The transformation attempts to suggest meaningful names, based on how the selected expressions appear in the program. For Example 1 from §2, because the numbers `120` and `80` are passed as the fourth and fifth arguments, respectively, to the function (`def rect (\(fill x y w h) ...)`), the suggested names include `w` and `h`. The user is asked to choose a name. The value itself (in this case, `120` or `80`) is not as important—the intention is that the values vary at once by a single change—so, to keep the number of `Results` small, the transformation does not ask the user to choose which value to use for the variable.

Move Definitions. Because of nested scopes and simultaneous bindings (*i.e.* tuples), there are many stylistic choices about variable definitions when programming in functional languages. The *Move Definitions* transformation takes a set of selected definitions and a single target position, and attempts to move the definitions to the target position. If the target position is before an expression, a new `let`-binding is added to surround the target. Whitespace is added or removed to match the indentation of the target scope. If the target position already defines a list pattern, then the selected definitions are inserted into the list. If the target position defines a single variable, then a list pattern is created. In cases where the intended transformation would capture variable uses or move definitions above their dependencies (errors that are easy to make when using text-edits alone), the transformation makes secondary edits (alpha-renaming variables and moving dependencies) to the program to avoid these issues. Our implementation of *Move Definitions* also provides options for whether or not to collapse multiple definitions into a single tuple, and also provides support for rewriting arithmetic expression definitions as an alternative way to deal with dependency inversion issues.

4 USER STUDY

We designed and implemented Deuce with the goal to incorporate structured editing within a text-based program editor. In this section, we describe a user study designed to measure the degree to which we were successful.

Besides the two novel mechanisms in our user interface design—structural code selection and context-sensitive preview menus—that we wish to evaluate, there are several additional factors at play. First, many users may not have extensive experience with functional programming languages, especially the custom Little language supported in our implementation. Second, our implementation provides some familiar transformations but some—particularly those involving target positions—are not. Furthermore, some users may prefer to use text-editing rather than structured edits, even when the latter can be used. These factors make it hard to perform a direct comparison between our implementation of Deuce and an existing system, such as Eclipse.

To mitigate these factors, we designed a study that compared Deuce with a "baseline" version of the system, with features designed to emulate the traditional text-select-based interface described in § 1. We then designed tasks, to be completed in both versions and *without* text-edits, to measure the effect of the new Deuce user interface features compared to the baseline ones. Below, we describe the different configurations of our system, our study procedures, and our results.

4.1 System Configurations

Recall that tools may be `Active` or `NotYetActive` based on one or more selected items and target positions (Figure 2).

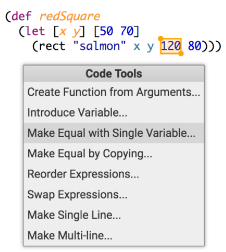
Traditional Mode ("Text-Select Mode"). To form the traditional mode of the tool, which we called Text-Select Mode in the user study materials, we implemented four interactions separate from the workflow described in §2 and §3.1 to invoke code tools.

(A) Code Tools Menu. The editor displays a Code Tools menu at the top of the window with a list of all transformations available in the system; this menu is akin to the Source and Refactor menus in Eclipse. The user selects a tool from this menu without first selecting anything in the program. Then, the editor displays a Tool Configuration Panel that displays tool-specific instructions. Tool Configuration Panels, which appear in all four interactions of Traditional Mode, are discussed below.

(B) Text-Select Single Argument + Code Tools Menu. This interaction is like Interaction A, except that the user first text-selects an item or target in the code. Like Eclipse, text-selecting requires the entire item to be selected, possibly with trailing or leading whitespace. Our implementation provides more generous text-selection mechanisms (*e.g.* largest containing expression, smallest surrounding expression), but the stricter version is used in the study because it is more similar to existing approaches [Murphy-Hill and Black 2008]. Also like Eclipse, all tools are displayed and enabled in the Code Tools menu, even if the tool is `Inactive` based on the selection.

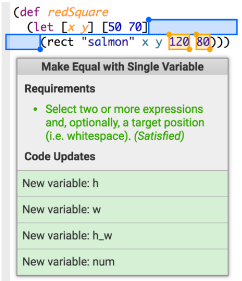
(C) Text-Select Single Argument + Right-Click Menu. After first text-selecting an item, as in Interaction B, the user right-clicks to trigger a pop-up menu that displays *only* plausible tools (`Active` or `NotYetActive`). A similar workflow is provided by Eclipse.

Returning to Example 1 from § 2, the screenshot on the right shows the right-click menu after text-selecting the `120` constant. By comparison, notice how this right-click menu displays more tools (`NotYetActive` tools in addition to `Active` ones). After the text selection is made, the editor draws an orange box (as with Deuce widgets) to identify the selection.



(D) Cursor-Select Single Argument + Right-Click Menu. For atomic code items (*i.e.* constants and variables), the user implicitly selects the item by right-clicking on the token (rather than text-selecting it) to trigger the right-click menu, as in Interaction C. Again, a similar workflow is provided by Eclipse.

Tool Configuration Panels. Each of the four interactions above trigger Tool Configuration Panels, which display the `requirements` string that explains how to invoke the tool. The user selects any additional arguments by hovering over and clicking structural selection widgets. That is, structural selection widgets are *not* accessible to make the primary selection, but they *are* used to make all remaining selections in a Tool Configuration Panel. The screenshot above shows the Configuration Panel after text-selecting `120`, as above, and then selecting `80` and a target position using structural selection. Because the tool requirements are satisfied, the panel displays the list of `Results`, each of which can be hovered to preview the change before selecting it.



Deuce Mode ("Box-Select Mode"). This configuration, called Box-Select Mode in the user study materials, isolates the new Deuce features. To review, the user holds down the Shift key, then hovers over and clicks one or more structural code selection widgets. When at least one widget is selected, the pop-up preview menu displays the list of `Active` tools.

There is no Code Tools menu at the top of the editor in this mode, even though the "full" version of our tool (not used by participants in the study) does; the list of tool names and descriptions in Tool Configuration Panels (which are *not* accessible in Deuce Mode) can help understand unfamiliar transformations.

Combined Mode. Our last configuration combines Traditional and Deuce Modes, with all interactions described above.

4.2 Questions and Procedures

We sought to address several questions:

- Is either mode more effective for (a) completing tasks, (b) rapid editing, or (c) achieving more with fewer transforms?
- Is either mode preferred by users? In which cases?

To answer these questions, we designed the following IRB-approved, controlled user study with 21 undergraduate and graduate students from the University of Chicago. We recruited users by sending emails to public mailing lists, offering a monetary incentive of $50 for participating in the two-hour study. Prior experience with functional programming or Sketch-n-Sketch was not required. Each user attended an individual session and was given the option to use the laptop and mouse provided by us or their own devices.

The primary components of the study included a tutorial portion followed by a tasks portion. We configured a pared-down version of the system that turned off all Sketch-n-Sketch features unrelated to the interactions being studied. The tutorial and tasks were set up as a self-guided progression of steps through the tool, to be completed at the user's own pace. In the description of the tutorial and tasks below, all random choices were made independently of other choices, as well as across users.

Our system logged user events to analyze the tutorial and tasks. We also recorded video of the users performing the tasks, for manual inspection in situations where the log information was insufficient or more difficult to process. Besides helping to get started and correct minor issues unrelated to Deuce, the user study proctor did not answer any questions about Deuce or the tasks. To wrap up, users answered questions about their programming background and experience using Deuce in an exit survey.

Tutorial. The first part of the tutorial introduced ordinary text-based programming in Little, emphasizing that the syntax would not be too important for subsequent tasks.

The majority of the tutorial introduced the code tools using both Traditional and Deuce Modes. The first tool introduced—*Rename Variable*, a familiar tool to many—was explained using all five interaction modes. But because the four interactions in Traditional Mode are largely similar, all subsequent tools introduced in the tutorial had only one set of instructions for Traditional Mode. For all tools introduced, a random choice was used to determine whether to explain Traditional or Deuce Mode first. In total, 10 of the 22 code tools in our implementation were demonstrated in the tutorial.

Table 1: Overview of the four head-to-head and two open-ended tasks. #LOC is non-blank lines of code in the starting program.

Name	#LOC	#Transforms	Example Tool Sequence (with minimum number of transforms required)
One Rectangle	9	3	Swap Expressions; Move Definition; Swap Definitions
Two Circles	11	2	Create Function from Definition; Reorder Arguments
Three Rectangles	11	2	Creating Function by Merging Definitions; Rename
Four Rings	7	4	Remove Argument; Rename; Move Definition; Add Arguments
Four Squares	9	7	Create Function by Merging Definitions; Create Function from Arguments; Rename (5x)
Lambda Icon	10	8	Make Equal with Single Variable (6x); Introduce Variable; Rename

To give a flavor of the tutorial, Example 1 in § 2 is adapted from the steps that introduced the *Make Equal* tools. In addition to tool-specific tutorial steps, we also dedicated a step for more practice with target positions, independent of a specific tool, because the notion of target positions was likely to be unfamiliar.

Tasks. After the tutorial, users worked on six *tasks*, each a different program and a list of one or more edits to perform using code tools. For some tasks, there were multiple different sequences of code tool invocations that could lead to the desired result. The starting programs ranged from 7 to 11 lines of code and required between 2 and 8 tool invocations (at minimum) to finish the tasks. Table 1 outlines the tasks. The Two Circles task was presented as Example 2 in §2. Extended task descriptions can be found in the Supplementary Appendices [Hempel et al. 2018].

Before every task, the participant was given a read-only reading period to understand the program before seeing the list of edits to perform. To emulate a real-world scenario where the programmer knows what to accomplish but may not quite remember all the steps, the task directions were written in a more natural style without direct reference to tool names—for example, "move the `ring` definition inside `target`" instead of "invoke *Move Definition* on the `ring` definition with a target position inside `target`."

Each of the first four tasks ("head-to-head tasks") was performed twice, once each in Traditional and Deuce Modes, resulting in eight *trials*. The first four trials comprised each of the four tasks, in random order and with one of the modes randomly chosen per trial. For the next four trials, the order of tasks was, again, randomized, each using the mode not chosen for the task in the first round. After these eight trials, the user performed each of the last two tasks ("open-ended tasks") once using the Combined Mode—both Traditional and Deuce Modes were available for use, to mix-and-match the two modes however they saw fit.

For each task, comments showed what the desired final code should look like, sometimes modulo minor whitespace differences. The editor provided an indicator about whether the task was completed, giving the user the option to Give Up at any point if needed. There was also a maximum time limit of six and twelve minutes for each head-to-head and open-ended task, respectively, with no indication about the time limit until and unless the user reached the two and four minutes remaining mark, respectively.

4.3 Results

Participants reported between 2 and 10 years of programming experience (mean: 5.1), of which between 0 and 3 years involved functional programming (mean: 0.76). 10 participants (48%) reported no prior functional programming experience. 8 participants reported using tools that supported automated refactoring (Eclipse, IntelliJ, and PyCharm all received multiple mentions). 4 participants reported some prior exposure to previous versions of the Sketch-n-Sketch project, but none reported knowledge of the code tools presented in the study.

For the study itself, 8 users brought their own laptop, the remaining 13 used ours. 15 participants used a mouse, and 6 relied on their laptop's trackpad. Each session took a mean of 1hr 44min (range: 1h 11m – 2h 27m). Users spent between 23 and 66 minutes on the tutorial (mean: 41) and 20 and 65 minutes on the tasks (mean: 44). The remaining time was spent on introductory remarks and the exit survey. All users attempted all tasks. Two trials were discarded because of tool malfunction, for a final total of 166 head-to-head trials and 42 open-ended tasks suitable for analysis.

The tasks proved moderately difficult. On average, each participant successfully completed 71% of the trials and open-ended tasks within the time limits, with 3 users completing them all and 1 user failing to complete any. Figure 4 shows completion rates by task. The One Rectangle and Lambda tasks had particularly low completion rates. Based on videos of failed attempts, many users struggled with choosing appropriate tools—*e.g.* many chose *Introduce Variables* rather than *Make Equal*, and some chose *Inline* rather than *Move Definitions* in an attempt to create a tuple definition. The tutorial was not sufficient for everyone to remember and understand all the tools needed for the tasks. The task descriptions may have also presented obstacles—*e.g.* for Lambda, the phrase "Define and use...", along with `(def [x y w h] ...)` in the final code, may have led some to use *Introduce Variables*, which would then require several roundabout transformations to complete the task. We believe these difficulties are largely independent of the user interface features. We now address each of the research questions in turn.

Is either mode more effective for completing tasks?. Figure 5 breaks down completion rates for head-to-head tasks by mode. Because each was attempted twice, to assess possible learning effects from already completing a task in the other mode, Figure 5 also differentiates between the user's first or second encounter with each task. Visually, the data suggest that on the first encounter with a task, Traditional Mode may better facilitate completion, and is also a better teacher for the subsequent encounter with Deuce Mode. In contrast, a first encounter with Deuce Mode may be less helpful for the second encounter with Traditional Mode.

To control for learning effects, a mixed effects logistic regression model [Gelman and Hill 2007] was fit with `lme4` [Bates et al. 2015] to predict task completion probability based upon fixed effect

Figure 4: Task completion rates pooled over both modes.

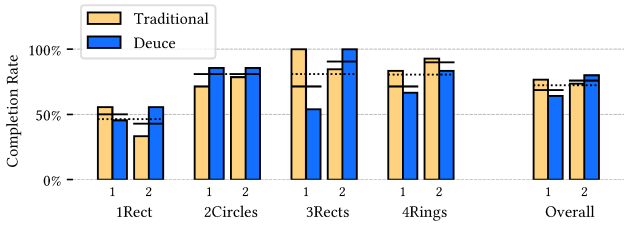


Figure 5: Head-to-head task completion rates by mode and by subject's first/second encounter with task. Overlaid lines indicated pooled completion rates.

predictors for the mode (coded as 0 or 1), the trial number (1-8), whether the trial was the second encounter with the task (0 or 1), whether the participant used a mouse (0 or 1), whether the participant used their own computer (0 or 1), and the interaction of mode with the second encounter (0, or 1 when Deuce Mode and a second encounter). To model differences in user skill and task difficulty, a random effect was added for each participant as well as each task, and a random interaction was added to model differences in the second encounter difficulty per task. Reported p-values are based on Wald Z-statistics.

In the fit model, the coefficient for mode was on the edge of significance (p=0.057), indicating that Traditional Mode did better facilitate task completion on the first encounter with a task. Given this, Deuce Mode performed better than expected on the second encounter (interaction term p=0.036), but not enough to confidently say that Deuce Mode was absolutely better than Traditional Mode for the second encounter (p=0.17). No other fixed effect coefficients approached significance.

Deuce Mode therefore seems to present a learning curve, but may be just as effective as Traditional Mode once that learning curve is overcome. This interpretation accords with the surveys: 5 participants wrote that Traditional Mode might be better for learning, and 4 participants—including 3 of the previous 5—said Deuce Mode was better when they knew the desired transformation. However, the data may be alternatively explained if Deuce Mode on the first encounter is a poor teacher, actively misleading users on the second encounter with Traditional Mode.

Is either mode more effective for rapid editing? Among trials successfully completed, the duration of each trial was measured from the start of configuration of the first refactoring to the end of the final refactoring. The distribution of these timings is presented in Figure 6, scaled relative to the mean duration for each task.

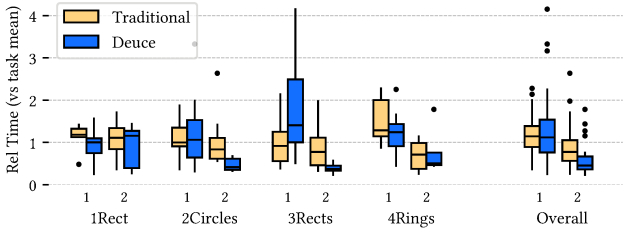


Figure 6: Head-to-head task durations for successfully completed trials, scaled relative to the mean time per task.

Again, to tease out if any of these differences are significant, from the same predictors described above two linear mixed effects models were fit to predict (1) trial duration and (2) the logarithm of trial duration (*i.e.* considering effects to be multiplicative rather than additive). Percentile bootstrap p-values for the fixed effect coefficients were calculated from 10,000 parametric simulate-refit samples.[1] For the first encounter with a task, Traditional Mode was insignificantly faster (by 13 seconds, p=0.44; or 9.2%, p=0.52). However, Deuce Mode was on average 25 seconds (p=0.13) or 36% (p<0.01) faster for the second encounter with a task, suggesting that Deuce Mode may be faster once users become familiar with the available tools. Most of the gain comes from less time spent in configuration—after discounting all idle thinking time between configurations, the model still reveals an 18 second difference.

Is either mode more effective for achieving more with fewer transforms? To determine if either mode facilitated more efficient use of interactions, the same mixed effects model was fit to predict the number of refactorings invoked during each successful trial, as well as the number of Undos. On the first encounter with a task, Traditional Mode accounted for an average of 2.0 fewer refactorings (p<0.01) and 2.1 fewer Undos (p<0.01), but on the second encounter no significant difference in number of refactorings or Undos was indicated. As a second encounter with Deuce Mode is faster than Traditional Mode, the speed gain thus appears to be explained by faster invocations rather than fewer invocations.

Is either mode preferred by users? In which cases? The two final open-ended tasks allowed participants to mix-and-match the two modes as they pleased. As shown in Figure 7, on both tasks the overwhelming number of users performed a greater share of refactorings using Deuce Mode. We believe a main advantage of Deuce Mode is that it simplifies the configuration of refactorings that require multiple arguments, as the user may select all the arguments together before choosing a transformation from a short menu. In Traditional Mode, the workflow is stuttered: the user must select a single argument, right-click to choose a transformation, then select the remaining arguments. However, for a refactoring requiring only a single argument, Traditional Mode is more streamlined: a user may simply select the desired transformation immediately after right-clicking on the first argument. Thus, for single-argument refactorings, Deuce Mode's advantages may be limited. A breakdown of mode usage by popular tools (Figure 8) lends support to

[1] See https://www.rdocumentation.org/packages/lme4/versions/1.1-13/topics/bootMer

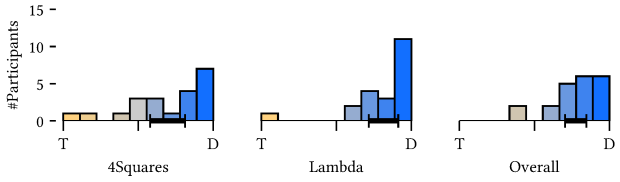


Figure 7: Distribution of user preferences for Traditional vs. Deuce Modes as measured by the ratio of refactorings performed by the user in each mode on the open-ended tasks. Far left represents all Traditional Mode refactorings; far-right indicates all Deuce Mode refactorings. The 95% confidence interval for the mean preference across all users is indicated (via percentile bootstrapping, 10,000 samples).

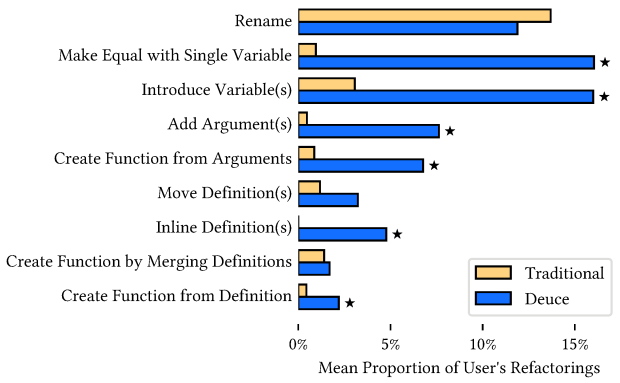


Figure 8: Mode usage for tools used by at least half of participants on the open-ended tasks. Deuce mode is preferred for most tools. Stars indicate differences significant at the 95% level (via percentile bootstrapping, 10,000 samples).

this hypothesis. For the most commonly used tool, *Rename*, which always takes only a single argument, participants used Traditional and Deuce Modes with roughly even frequency. Most other tools showed strong preferences towards Deuce Mode, with the notable exception of *Create Function by Merging Definitions*. Because the Four Squares task required invoking this tool with four expressions, according to the hypothesis, users should prefer Deuce Mode. The videos revealed that several users were unable to discover how to structurally select a function call, which required hovering on the open parenthesis (not demonstrated in the tutorial). Several of these users were, however, able to invoke the tool by text-selecting a function call or by starting from the full Code Tools menu.

Subjectively, the concluding survey asked whether Deuce or Traditional Mode worked better for each head-to-head task, measured on a 5-point scale from "Text-Select Mode worked much better" to "Box-Select Mode worked much better". For each participant, a random choice determined which mode appeared at each end of the scale. As shown in Figure 9, on average a similar modest preference for Deuce Mode was expressed for each task.

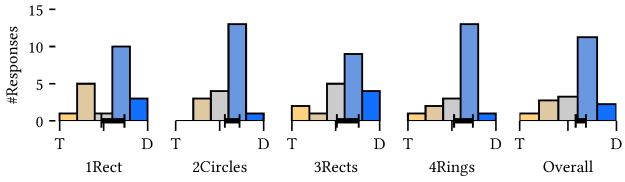


Figure 9: Surveyed subjective preference for Traditional vs. Deuce Modes for the head-to-head tasks. The 95% confidence interval for the mean preference across all users is indicated (via percentile bootstrapping, 10,000 samples).

On the free-response portion of the survey, several explanations were given for this preference for Deuce Mode. 3 participants appreciated the ability to select multiple arguments; 2 other participants appreciated selecting all arguments before selecting a tool; 1 other participant appreciated the smaller menu of refactorings; and 1 other participant appreciated the ease of starting a refactoring by clicking code objects rather than having to create a text selection.

Altogether, users demonstrated a strong objective and modest subjective preference for Deuce over Traditional Mode, suggesting that Deuce accomplishes its goal to provide a more human-friendly interface to identify, configure, and invoke refactorings.

Limitations. There are several threats to the validity of our experimental setup. One is that our emulation of traditional features may have been less effective than those features in existing tools. Another is that the participants may have felt compelled to use Deuce Mode (which could likely have been deduced to be more novel than Traditional Mode) more during the open-ended tasks—and pronounce a preference for it in the survey—because the participants were drawn from the same academic community as the authors. Another is that participants used the tool in heterogeneous environments—different computers and browsers, configured with different screen sizes and mouse settings. Performance on the tasks may have also been affected by the presence of the user study proctor and video recording device. According to self-reported assessments, participants were relatively unfamiliar with functional programming and with refactoring tools, so the results may differ for users with more extensive experience. Finally, our results were obtained on small programs and tasks in a prototype language.

Future Improvements. There are opportunities to improve our implementation of Deuce. First, to reduce the learning curve, it would be worth adding more explanatory features (*e.g.* in a tutorial, or within the tool when the user selects certain kinds of items for the first time), particularly for unfamiliar transformations (*e.g. Move Definitions*) and for unfamiliar user interface features (*i.e.* target positions). Enabling the full Code Tools menu may also help because of the descriptions of requirements in the Tool Configuration Panels (*cf.* the "Deuce Mode" discussion). Also, to allow easy corrections of misconfigured refactorings, it would help if Undo restored the previous selection state rather than just the previous version of the code; we have since implemented this feature.

5 RELATED WORK

We describe the most closely related ideas in structured editing and refactoring. Ko and Myers [2006], Lee et al. [2013], and Omar et al. [2017] provide more thorough introductions.

5.1 User Interfaces for Structured Editing

Compared to traditional text-selection and menus, several alternative user interface features have been proposed to integrate structured editing more seamlessly within the text-editing workflow.

Text Selection. Murphy-Hill and Black [2008] identify that text selection-based refactoring is prone to error, particularly for statements that span multiple lines and that have irregular formatting. They propose two prototype user interface mechanisms, called Selection Assist and Box View, to help. With Selection Assist, the user positions the cursor at the start of a statement, and the entire statement is highlighted green to show what must be selected (using normal text-selection). With Box View, the editor draws a separate panel (next to the code editor) that shows the tree structure of the program with nested boxes. When selecting text in the editor, the nested boxes are colored according to which code items are completely selected. Similarly, the user can select a nested box in the Box View to select the corresponding text in the code.

In contrast, our structural selection polygons are drawn directly atop the code, at once helping to identify (like Box View) and select (like Selection Assist), which aims to mitigate the context switching overhead of Box View identified by Murphy-Hill and Black [2008].

Drag-and-Drop Refactoring. Lee et al. [2013] propose a tool called DNDRefactoring that eliminates the use of menus altogether. They demonstrate how many common Eclipse refactorings can be unambiguously invoked with a drag-and-drop gesture without the need for any additional configuration. This is a compelling workflow for situations in which the user can (a) readily *identify* an intended refactoring based on a preconceived notion (*e.g.* its name), (b) unambiguously *invoke* the intended refactoring by a single-source, single-target drag-and-drop gesture, and (c) accept the *default configuration* of the refactoring. It would be useful to add drag-and-drop gestures to Deuce for transformations that satisfy these three conditions. However, our user interface supports situations when one or more of these three conditions fails to hold.

Hybrid Editors. Compared to "fully" structured editors, several *hybrid editor* approaches augment text-based programs with additional information. Barista [Ko and Myers 2006] is a hybrid Java editor where *structure views* can be implemented to present alternate representations of structural items instead of text. For example, an arithmetic expression may be rendered with mathematical symbols, a method may be accompanied by interactive documentation with input-output examples, and structures may be selectively collapsed, expanded, or zoomed. Omar et al. [2012] introduce a similar notion to structure views, called *palettes*, where custom displays can be incorporated based on the type of a subexpression. For example, a color palette can provide visual previews of different candidate color values, and a regular expression palette can show input-output examples for different candidate regular expressions. In Greenfoot [Brown et al. 2016], program text is separated into structural regions called *frames*, which are created and manipulated with text- and mouse-based operations that are orthogonal to the text-edits within a frame. Code Bubbles [Bragdon et al. 2010] allows text fragments to be organized into *working sets*, which are collections of code, documentation, and notes from multiple files that can be organized in a flexible way. Outside of the views, palettes, frames, and working sets in the above hybrid editors, the user has access to normal text-editing tools.

Our approach is complementary to all of the above: in places where code fragments—regardless of their granularity and their relationship to alternative or additional pieces of information—are represented in plain text, we aim for a lightweight user interface to structurally manipulate it.

Refactoring with Synthesis. In contrast to direct manipulation in DNDRefactoring and Deuce, Raychev et al. [2013] propose a workflow where the user starts a refactoring with text-edits—providing some of the changes after the refactoring—and then asks the tool to synthesize a sequence of refactorings that complete the task. This text-based interface and the mouse-based interfaces of DNDRefactoring and Deuce are complementary.

5.2 Program Transformations

Automated support for refactoring [Fowler 1999; Griswold 1991; Roberts et al. 1997] has been aimed primarily at programs written in class-based, object-oriented languages.

Refactoring for Functional Languages. HaRe [Brown 2008; Li 2006; Thompson and Li 2013] is a refactoring tool for functional languages, such as Haskell, where features—including first-class functions (*i.e.* lambdas), local bindings, tuples, algebraic datatypes, and type polymorphism—lead to editing tasks that are different from those supported in most typical refactoring tools for object-oriented programs. Our user interface could be incorporated by HaRe to expose the supported transformations with lightweight direct manipulation. HaRe provides a larger catalog of transformations than our current implementation of Deuce. However, the details of our *Move Definitions* and *Make Equal* transformations are, to the best of our knowledge, not found in existing tools.

6 CONCLUSION

Based on our experience and the results of our user study, we believe Deuce represents a proof-of-concept for how to achieve a lightweight, integrated combination of text- and structured editing. In future work, our design may be adapted and implemented for full-featured programming languages and development environments, incorporating additional well-known transformations (*e.g.* Fowler [1999]; Thompson and Li [2013]). Additional direct code manipulation gestures, as well as incremental parsing (*e.g.* the algorithm of Wagner and Graham [1998] used by Barista [Ko and Myers 2006]), could further help streamline, and augment, support for structured editing within an unrestricted text-editing workflow.

ACKNOWLEDGMENTS

The authors thank Shan Lu, Elena Glassman, Aaron Elmore, Peter Scherpelz, and Blase Ur for suggestions about this paper. This work was supported by National Science Foundation Grant No. 1651794, and a University of Chicago Liew Family Research Fellows Grant.

REFERENCES

Amjad Altadmri, Michael Kölling, and Neil Christopher Charles Brown. 2016. The Cost of Syntax and How to Avoid It: Text versus Frame-Based Editing. In *Computer Software and Applications Conference (COMPSAC)*.

Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. 2015. Fitting Linear Mixed-Effects Models Using lme4. *Journal of Statistical Software* (2015).

Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. 2010. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In *International Conference on Software Engineering (ICSE)*.

Christopher Brown. 2008. *Tool Support for Refactoring Haskell Programs.* Ph.D. Dissertation. University of Kent.

Neil Christopher Charles Brown, Amjad Altadmri, and Michael Kölling. 2016. Frame-Based Editing: Combining the Best of Blocks and Text Programming. In *Conference on Learning and Teaching in Computing and Engineering (LaTiCE)*.

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Conference on Programming Language Design and Implementation (PLDI)*.

Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc.

Andrew Gelman and Jennifer Hill. 2007. Data Analysis Using Regression and Multilevel/Hierarchical Models. (2007).

William G. Griswold. 1991. *Program Restructuring as an Aid to Software Maintenance.* Ph.D. Dissertation. University of Washington.

Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Symposium on User Interface Software and Technology (UIST)*.

Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: A Lightweight User Interface for Structured Editing. (2018). Extended version of ICSE 2018 paper available as *CoRR abs/1707.00015*.

Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. 2005. Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing. In *Human Factors in Computing Systems (CHI)*.

Andrew J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Human Factors in Computing Systems (CHI)*.

Yun Young Lee, Nicholas Chen, and Ralph E. Johnson. 2013. Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation. In *International Conference on Software Engineering (ICSE)*.

Huiqing Li. 2006. *Refactoring Haskell Programs.* Ph.D. Dissertation. University of Kent.

John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *Transactions on Computing Education (TOCE)* (2010).

Erica Mealy, David Carrington, Paul Strooper, and Peta Wyeth. 2007. Improving Usability of Software Refactoring Tools. In *Australian Software Engineering Conference (ASWEC)*.

Jens Monig, Yoshiki Ohshima, and John Maloney. 2015. Blocks at Your Fingertips: Blurring the Line Between Blocks and Text in GP. In *IEEE Blocks and Beyond Workshop (BLOCKS AND BEYOND)*.

Emerson Murphy-Hill and Andrew P. Black. 2008. Breaking the Barriers to Successful Refactoring. In *International Conference on Software Engineering (ICSE)*.

Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *International Conference on Software Engineering (ICSE)*.

Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *Summit on Advances in Programming Languages (SNAPL)*.

Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active Code Completion. In *International Conference on Software Engineering (ICSE)*.

Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin Vechev. 2013. Refactoring with Synthesis. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Christoph Reichenbach, Devin Coughlin, and Amer Diwan. 2009. Program Metamorphosis. In *European Conference on Object-Oriented Programming (ECOOP)*.

Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Communications of the ACM (CACM)* (2009).

Don Roberts, John Brant, and Ralph Johnson. 1997. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems* (1997).

Friedrich Steimann and Jens von Pilgrim. 2012. Refactorings Without Names. In *International Conference on Automated Software Engineering (ASE)*.

Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* (1981).

Simon Thompson and Huiqing Li. 2013. Refactoring Tools for Functional Languages. *Journal of Functional Programming* (2013).

Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, Manuel Fahndrich, and Sebastian Burckhardt. 2012. TouchDevelop: App Development on Mobile Devices. In *International Symposium on the Foundations of Software Engineering (FSE)*.

Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. 2012. Use, Disuse, and Misuse of Automated Refactorings. In *International Conference on Software Engineering (ICSE)*.

Tim A. Wagner and Susan L. Graham. 1998. Efficient and Flexible Incremental Parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1998).