reCode: A Lightweight Find-and-Replace Interaction in the IDE for Transforming Code by Example

Wode Ni
woden@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, USA

Joshua Sunshine
sunshine@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, USA

Vu Le
levu@microsoft.com
Microsoft
Redmond, USA

Sumit Gulwani
sumitg@microsoft.com
Microsoft
Redmond, USA

Titus Barik
tbarik@microsoft.com
Microsoft
Redmond, USA

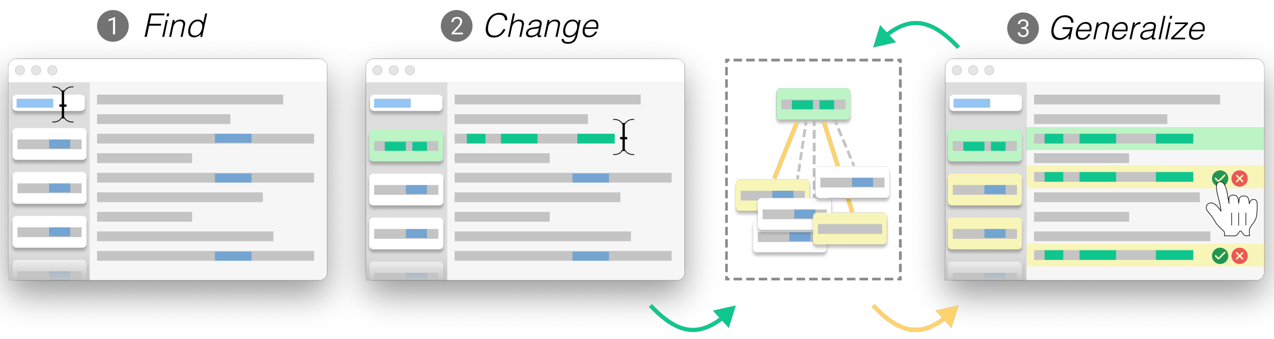


Figure 1: reCode is a mixed-initiative tool that automates code transformations via an example-driven interaction. The developer **finds** relevant locations in their codebase and directly perform **changes** inline. Based on the search results and user edits, reCode automatically **generalizes** edits to other applicable locations as the developer iteratively refines code changes.

ABSTRACT

Software developers frequently confront a recurring challenge of making code transformations—similar but not entirely identical code changes in many places—in their integrated development environments. Through formative interviews ($n = 7$), we found that developers were aware of many tools intended to help with code transformations, but often made their changes manually because these tools required too much expertise or effort to be able to use effectively. To address these needs, we built an extension for Visual Studio Code, called reCode. reCode improves the familiar find-and-replace experience by allowing the developer to specify a straightforward search term to identify relevant locations, and then demonstrate their intended changes by simply typing a change directly in the editor. Using programming by example, reCode automatically learns a more general code transformation and displays these transformations as before-and-after differences inline, with clickable actions to interactively accept, reject, or refine the proposed changes. In our usability evaluation ($n = 12$), developers reported that this mixed-initiative, example-driven experience is intuitive, complements their existing workflow, and offers a unified approach to conveniently tackle a variety of common yet frustrating scenarios for code transformations.

KEYWORDS

code transformation, program synthesis, find-and-replace

ACM Reference Format:
Wode Ni, Joshua Sunshine, Vu Le, Sumit Gulwani, and Titus Barik. 2021. reCode: A Lightweight Find-and-Replace Interaction in the IDE for Transforming Code by Example. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21), October 10–14, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3472749.3474748

UIST '21, October 10–14, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8635-7/21/10.
https://doi.org/10.1145/3472749.3474748

1 INTRODUCTION

Maria, a front-end developer, wants to rewrite the visual styles in her project to use vanilla CSS instead of the current styled-components library. In other words, she wants to find lines of code in her project that look like this:

```
border: 1px solid ${props => props.theme.black};
```

and replace them to look like this:

```
border: 1px solid var(--black);
```

To estimate the scope of this task, Maria invokes the find interface in her IDE and searches for `props.theme`. The interface returns around 30 results, scattered across multiple files. How should Maria complete the task?

Developers like Maria frequently run into these kinds of systematic, repetitive *code transformations*—similar but not entirely identical code changes in many places [23, 41, 42]. If it turns out there are only a few lines of code to edit, Maria could simply make the replacements manually in her IDE. If there are thousands of lines to edit, however, manual approaches become intractable. Then, there are a bewildering array of tools for developers to turn to for automation. A common option is to write regular expressions, which are essentially sequences of characters that specify search patterns. More elaborate approaches include text-based find-and-replace tools like `sed` [28] or `ripgrep` [6], or language-aware tools like structural find-and-replace [36] and `jscodeshift` [3].

More often than not, developers end up in an unpleasant "murky middle" that is somewhere between these two extremes. In this murky middle, manually making the changes is both time consuming and error prone, yet the investment required to automate with a regular expression or script is also unappealing and difficult even for seasoned developers [32]—it is possible that automating would take longer than doing the task manually. Neither strategy feels "just right."

Through formative interviews with developers, we identified limitations in current code transformation tools that were barriers to developers. First, developers struggled to decide between transforming code manually versus investing in using a tool to automate the task, particularly when there are a murky middle number of edits to make. Second, developers reported that writing code transformation scripts was complicated because of the many edge cases that arise. Third, scripting approaches were often too monolithic, requiring developers to make code transformations in bulk across their entire project. This made it difficult for developers to reason about how the code transformation impacts their code. In short, developers desired a more incremental and interactive approach that allowed for automation while still allowing for oversight and occasional intervention.

To address these needs, we propose a mixed-initiative [19] tool, called reCode, that offers developers a lightweight interaction for transforming code while balancing automation and inspection. reCode is implemented as a Visual Studio Code extension, and enhances the familiar find-and-replace experience. reCode users first specify a straightforward search term to identify relevant locations of interest for the code transformation. To remove the burden of having to write a complicated regular expressions or script, developers demonstrate their intended code transformation to reCode by simply typing the code change directly in their editor. reCode leverages programming-by-example to automatically learn a more general code transformation, across a variety of transformation scenarios. reCode displays these additional transformations as before-and-after differences inline, and offers the developer clickable actions through which they can interactively accept, reject, or refine the proposed transformations.

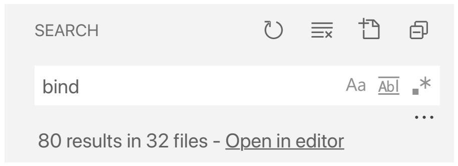
The contributions of this paper are as follows:

- We propose a mixed-initiative interaction for the IDE that improves the familiar find-and-replace experience through programming-by-example. This interaction removes the need to need write regular expressions or other complicated scripts for a variety of code transformations. We implement this interaction as an extension, called reCode, for Visual Studio Code.
- reCode implements a feedback-driven, semi-supervised program synthesis technique, called ReFazer* [16]. ReFazer* accepts tree-based input and output examples to learn program transformations. reCode surfaces this technique as a usable system.
- Through a usability evaluation with 12 developers, we demonstrate that the reCode example-driven experience is intuitive, complements their existing workflow, and offers a unified approach to conveniently tackle a variety of common yet frustrating scenarios for code transformations.

2 A DEMO OF RECODE

Maria used reCode to rewrite her visual styles based on a colleague's recommendation: *"it's like find and replace. Just start editing after you find things and it'll do the rest."* The next day, she decides to tackle a more complex clean-up task. Her application uses React and was originally written in JavaScript ES5. The application had many `bind` calls in class constructors. These `bind` calls were needed in ES5 to allow methods to work as they do in other languages. With the new version of JavaScript, these calls are no longer required[1]. Maria simplifies her code by: (1) deleting all lines that look like `this.func = this.func.bind(this)` and (2) rewriting the corresponding method declarations as "arrow functions".

To see how many of these functions there are, Maria types `bind` in the Search Box (Figure 2 B) to search in the repository. Maria thinks, "it's 4 PM now and I want to get this done soon. If there are only three of these functions, I'll just do them manually." Unfortunately, the Summary View (Figure 2 A) shows 80 matches spread across 32 files!

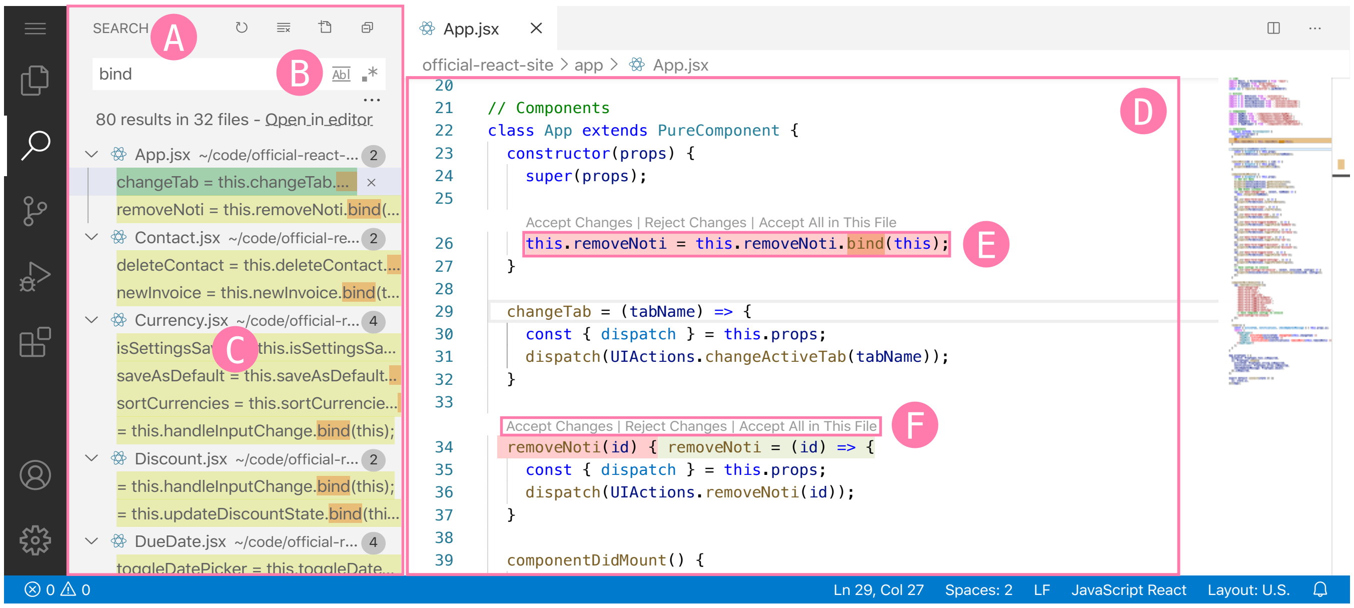


bind

80 results in 32 files - Open in editor

Behind the scenes, Maria's initial `bind` search with find-and-replace has already activated the reCode tool. She clicks on the first result in `App.jsx` and starts to edit the relevant lines for the `changeTab` function. She removed the `this.changeTab.bind(this)` call from line 25 and added `=` before `(tabName)` and `=>` after on line 29:

[1] https://reactjs.org/docs/faq-functions.html#how-do-i-bind-a-function-to-a-component-instance

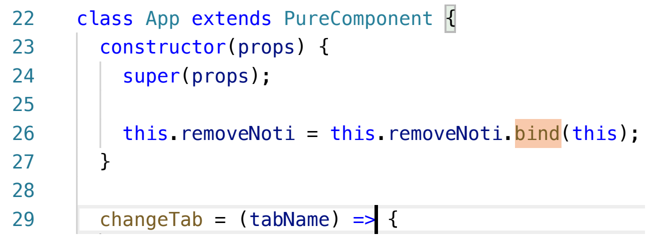


```
SEARCH
bind
80 results in 32 files - Open in editor
App.jsx ~/code/official-react-... 2
  changeTab = this.changeTab....
  removeNoti = this.removeNoti.bind(...
Contact.jsx ~/code/official-re... 2
  deleteContact = this.deleteContact...
  newInvoice = this.newInvoice.bind(t...
Currency.jsx ~/code/official-r... 4
  isSettingsSa    this.isSettingsSa...
  saveAsDefault = this.saveAsDefault...
  sortCurrencies = this.sortCurrencie...
  = this.handleInputChange.bind(this);
Discount.jsx ~/code/official-r... 2
  = this.handleInputChange.bind(this);
  = this.updateDiscountState.bind(thi...
DueDate.jsx ~/code/official-r... 4
  toggleDatePicker = this.toggleDate...
```

```
App.jsx
official-react-site > app > App.jsx

// Components
class App extends PureComponent {
  constructor(props) {
    super(props);

    Accept Changes | Reject Changes | Accept All in This File
    this.removeNoti = this.removeNoti.bind(this);
  }

  changeTab = (tabName) => {
    const { dispatch } = this.props;
    dispatch(UIActions.changeActiveTab(tabName));
  }

Accept Changes | Reject Changes | Accept All in This File
  removeNoti(id) { removeNoti = (id) => {
    const { dispatch } = this.props;
    dispatch(UIActions.removeNoti(id));
  }

  componentDidMount() {
```

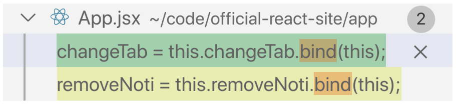
```
⊗ 0 ⚠ 0    Ln 29, Col 27   Spaces: 2   LF   JavaScript React   Layout: U.S.
```

Figure 2: The user interface of reCode. In the Summary View (A), the developer enters a simple query in the Search Box (B) and previews changes to match results. The developer's edits are highlighted in green and changes generalized from those edits are highlighted in yellow (C). The developer directly edits the source code in the main editor (D) and reCode synthesizes a generalized transformation. In the Inline Diff, suggested deletions are highlighted in pink and suggested replacements are highlighted in green. (E). The developer can accept or reject the suggestions via Inline Actions (F).



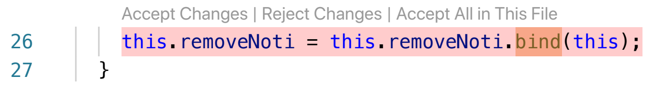
```
class App extends PureComponent {
  constructor(props) {
    super(props);

    this.removeNoti = this.removeNoti.bind(this);
  }

  changeTab = (tabName) => {
```

Maria notices in the search result for `changeTab` is now highlighted in green and she understands that reCode is generalizing her edits. Immediately after, other results light up in yellow, indicating suggestions made by reCode:



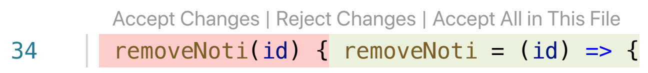
```
App.jsx ~/code/official-react-site/app 2
  changeTab = this.changeTab.bind(this);   ×
  removeNoti = this.removeNoti.bind(this);
```

Within the main editor pane (Figure 2 E), reCode gives two suggestions related to `removeNoti`. The first one seems correct: reCode proposes to delete the whole line with the `bind` call.



```
    Accept Changes | Reject Changes | Accept All in This File
    this.removeNoti = this.removeNoti.bind(this);
  }
```

Looking at the second suggestion, Maria concludes it's correct, too: the line with the `bind` call is removed, and reCode correctly kept the argument `id` for the function declaration (but if it hadn't been, Maria could have clicked "Reject Changes" to revert to the original or changed the code manually—reCode would learn from this correction and update its suggestions).



```
Accept Changes | Reject Changes | Accept All in This File
  removeNoti(id) { removeNoti = (id) => {
```

Maria clicks "Accept Changes" for both suggestions. She then clicks on several other results in the Summary View (Figure 2 A) to review the changes proposed by reCode. In the first three files, she clicks "Accept Changes" for each suggestion that she is confident about. To speed things up, she then goes to the rest of the files, review all of the changes, and clicks "Accept All in This File" once she determines everything is correct. Using this workflow, inspecting each file takes about 10 to 20 seconds, and she finishes changing all of her changes in under 10 minutes.

Now imagine doing the same task without reCode. Maria would have faced the same challenge of the "murky middle" described in Section 1. On the one hand, changing all 80 instances manually can easily take an hour and is error prone. On the other hand, it might not be worth the investment to write a custom script or complex regular expression to feed to a find-and-replace tool. For these reasons, Maria prefers the convenience of reCode to help her accomplish a variety of day-to-day code transformation tasks.

3 BACKGROUND AND RELATED WORK

The design of reCode is inspired by BluePencil [35], which implements a comparable underlying synthesis technology to reCode's engine [16], but surfaces the interaction through a different workflow: BluePencil *passively* detects and presents code transformation suggestions as "quick fix" lightbulbs to the developer as they edit their code, which the developer can either accept or ignore.

In contrast, reCode supports developers who frequently desire to have more control over their code transformations (Section 4).

The rest of this section describes related work on challenges developers have making code transformations and the multitude of programmatic approaches to code transformations.

3.1 Challenges of Making Code Transformations

Developers edit their code in a patterned and repetitive way to fix bugs [43], migrate from one API/language to another [11, 21], or make systematic changes to their codebases [22].

Nguyen et al. [41] conducted a large-scale study to show that 70-100% of small changes are repeated, and the repetitiveness of changes decreases exponentially as the change size increases. The smaller, fine-grained changes are especially meaningful and pervasive in both time and space: a given code change is often repeated by others, and the same developer has usually made the same kind of change in the past [42]. Within the same codebase, Kim et al. [23] found that “locally unfactorable, consistently changing clones” (that is, duplicated code that cannot be easily factored out and always change together) are common, and changing these clones together can be error-prone and difficult.

Automated tools aim to help developers make code transformations, but they are often too hard to use, leading to tool abandonment. For instance, Murphy-Hill et al. [38] found that 90% of changes that refactoring tools already support are performed without the help of tools. Most editors have find-and-replace functionalities baked in and support regular expressions for more general code transformations. However, find-and-replace can be error-prone [33] and regular expressions are especially hard to use [32].

reCode addresses the need for a more intuitive and reliable tool to automate repetitive code changes. It improves on the familiar find-and-replace user experience and leverages program synthesis to generate semantic code transformations from developers' direct edits.

3.2 Programmatic Approaches to Code Transformations

To automate code transformations, developers can write queries or scripts that typically operate in *batch* across their repository. These tools emphasize either text-level or tree-level transformations.

Text-based tools allow developers to perform changes to programs by matching a string pattern and treating the code as an unstructured *string of text*. Tools in this space include regular expressions [14] or regex-based codemods [1].

Instead of operating on strings, developers also can use tools that provide access to the code's abstract syntax tree (AST), types, or another language-specific information. Structural find-and-replace tools let developers transform their code by specifying patterns and grammatical constructs that take the code structure into account. For example, in these representations it becomes possible for the developer to specify constraints like “within class constructors only” or “fields of type integer.” JetBrain's family of IDEs supports structural find-and-replace for a variety of programming languages [36]. Comby [50] introduces a simpler query syntax for find-and-replace by generating parser combinators. Because Comby understands the syntax of code blocks, strings, and comments, Comby queries are usually more concise and readable than alternatives like regular expressions.

More elaborate code transformations require developers to go beyond queries and rewrite rules to scripts that directly operate on ASTs. `jscodeshift` transforms JavaScript programs via an API for JavaScript AST nodes. Rafazar includes a domain-specific language that encodes AST-level edit actions for program transformations [47]. Although these APIs may suffer from a variety of usability problems [40], AST transformers have shown their robustness and scalability for transforming ultra-large codebases [13, 51]. Refactorings tools [5, 7, 31, 37] are also instances of code transformation scripts, as are linters [2, 4].

Developers using reCode sidestep the decision of which approach to code transformation to choose. Developers edit examples and reCode “invisibly” [39] programs code transformations for them.

3.3 Editing by Example

In editors, programming-by-example systems infer changes to text or source code based on concrete user actions on the source text and/or other representations of the program. These inferred changes are often high-level programs consistent with the user actions but generalized to similar instances [18].

Several early programming-by-example tools, beginning in the 1980s, can operate on text [15, 25–27, 44, 45, 48, 49, 52], either by inferring a program from input-output examples (result-based) or recording users' edit steps (action-based). Nix synthesizes string transformation patterns from a set of input-output examples provided by the user [44]. The transformations are expressed as *gap programs*. SMARTEdit does not require an output example upfront and learns string-based macros from direct edits on an input example [25]. It requires an explicit start/stop command and treats all the text in the editor as the input example. Some editors allow developers to record edit steps as scripts called *keyboard macros*. For instance, both GNU Emacs [48] and `vi` [45] users can encode edit actions as a program and replay the same sequence of actions elsewhere. Because ordering is important in the edit steps, macros are known to be brittle and difficult to specify correctly [26, 49]. Different from text-based tools, reCode is tree-based and generates AST transformations. The resulting code transformations are resilient to edit ordering and formatting variations.

Sydit and Lase are Eclipse plugins for transforming functions or methods [29, 30]. In contrast to reCode's lightweight user experience which allows for fine-grained code transformations, Sydit requires developers to make code transformation at the method level. ReSynth is an Eclipse plugin that generates a sequence of refactoring operations from user edits [46]. reCode supports code transformations that are not limited to an existing catalog of refactorings.

Lapis [34] is a specialized editor that allows users to edit multiple lines of code concurrently. Lapis asks users to specify a natural-language like query to seed the examples. reCode differs in that it allows a straightforward, keyword-style search, and provides a more sophisticated synthesis backend. Codelink is an extension for XEmacs. The tool requires the developer to “link” code duplicates, or *code clones*, explicitly. Once these code clones are linked, any edits the developer makes simultaneously updates the other linked locations. In other words, Codelink's interaction is a variation of multi-cursor editing in modern editors [8, 9]. By contrast, reCode does not rely on cursor position and uses the developers' initial search term to bootstrap relevant locations. reCode presents a lightweight interaction workflow different from existing tools.

4 FORMATIVE INTERVIEWS AND DESIGN GOALS

To discover challenges that developers have with existing code transformation tools, we sent out an initial recruitment survey at a large software company to developers with at least three years of experience, sampled from their company address book. The survey pre-screened for prior experience using tools in participants' programming environments that allow them to perform code transformations, such as find-and-replace, refactoring, or other structural find-and-replace tools. We interviewed 7 of these survey respondents (F1-F7) to understand how they use tools to automate code editing. The interviews serve as a need-finding activity, from which we derive design goals for reCode.

In the interviews, we asked about the challenges they faced with code transformations, which tools they use to automate them, and why the tools they use work or don't work for them. Five participants showed us code samples from recent tasks, which we used to design tasks for our later usability evaluation (Section 6). From these interviews, we identified three common problems across participants.

First, developers reported having to make continuous trade-offs between performing code transformation tasks manually and reaching for programmatic approaches that enable automation, which resulted in decision fatigue. While some participants experimented with writing custom scripts (F1, F3) or regular expressions (F2, F4, F6, F7) to automate tasks, F2 reported encountering unanticipated barriers: "I could use find-and-replace, copy-paste, use multi-cursors, or use refactorings. But none of them worked exactly the way I wanted." Given these uncertainties, participants often impulsively gravitated towards more familiar, manual strategies like find-and-replace because it minimized their decision fatigue and was the path of least resistance (F1-F7). As F2 described, "there's probably already a secret tool or some magical trick [in my editor]. I just don't want to look."

Second, writing a robust regular expression or script is tricky, and several participants desired more lightweight but still expressive approaches. As one example, F7 described trying to use a regular expression but that the language lacked a "good way to specify context or scope." They also used macros, which were more generic but indicated that "the amount of time it takes to remember how to do macros doesn't justify using it for ordinary tasks" (F7). F1 added that when scripting, they "often run into these corner cases that the script doesn't handle" and wonders, "Do I really have to write my own static analyzer to do it correctly?" They desired an editor to "do it automagically, sees you doing this many times, and automates this." F6 explicitly described an example-driven experience: "I want to find-and-replace by example. I want to edit a file directly and say 'Apply that elsewhere' ".

Third and finally, participants reported a need for human oversight and inspection in automated approaches. F3 and others (F1, F2, F4, F5) worried about "over-replacing" and "matching on the wrong thing. Because things like find-and-replace are syntax-based, your compiler may not catch the error, so I have to check it manually." Several participants mentioned that 'Apply all' is "dangerous" (F1-F6) and F2 had to "watch very carefully to make sure I don't replace things I don't want." To guard against these issues, all participants shared their experiences building up search queries iteratively from a simple keyword, and then further narrowing down their results as needed.

Participants reported that their existing tools were mismatched with their desire for inspection. F3, for example, said that they "spend one hour to click apply and next and apply, and I'll just give up and apply all, hoping the compiler catches errors for me," and F5 reports that automated tools "stress me out and I don't really trust them" because they can't easily verify if the code transformations have been correctly applied. Towards improvements in the editor, F2 and F4 suggested "live previews and highlights" to inspect the changes within the editor. F5 indicated that comparisons in current editors are difficult because they use too little "screen real estate" and require them to refine code transformations through "tiny text boxes."

Based on their reported experiences and feedback, we reflected on their needs and formulated several design goals to address them:

- **D1. Provide a unified entry point for code transformations.** To minimize decision fatigue, developers should be able to make a variety of common code transformations through a familiar user experience.
- **D2. Offer a lightweight way to transform code.** Writing regular expressions and custom scripts are difficult. An intelligent user experience should provide this capability "automagically," offloading script building to the system.
- **D3. Design affordances that enable oversight and inspection for code transformations.** Developers were cautious about automated tools over-replacing or matching on the wrong things, and existing tools were mismatched with developer expectations. Developers should be able to incrementally inspect code transformations and more easily compare their results.

5 SYSTEM DESIGN AND IMPLEMENTATION

reCode realizes the design goals from Section 4 and offers a user experience that: 1) reduces the decision overhead of having to choose among different tools; instead, the developer can use find-and-replace as a unified entry-point for their code transformation (D1), 2) eliminates the burden of having to author complicated regular expressions or scripts; instead, the developer can directly type their change in the editor (D2), and 3) removes the requirement to inspect all code transformations in bulk; instead, the developer can incrementally inspect, apply, and revise their code transformations (D3).

reCode is implemented as an extension of Visual Studio Code (VSCode). Most of its features are implemented within the Search View and main editor pane. First, we augmented the Search View to indicate the state of each search match. Second, the main editor captures developer edit events and the ReFazer* synthesizer runs as an editor service in the background and generates transformation programs based on the edits. Finally, we implemented inline code diffs to surface the synthesizer's suggestions and we adapted Code Lenses[2] to allow the developers to interact with the synthesizer.

5.1 reCode Workflow

In this section, we will walk through the detailed design of reCode's mixed-initiative workflow (Figure 1), which we demonstrated in Section 2.

5.1.1 Step 1: Bootstrapping Synthesis via Find. The developer initiates the workflow by typing search terms in the Search Box (Figure 2 A). The search and results are displayed in the corresponding Result View (Figure 2 B). As we described in Section 4, our participants were comfortable with constructing search queries via simple keywords and narrowing down results using find. Consequently, we made an intentional design decision to sacrifice some expressiveness in search (for example reCode users cannot limit search to "only within fields of a class") to favor simplicity. reCode only supports conventional plain-text search.

Because the developer provides search terms that are less precise than the locations they actually intend to change, the search results will be a *superset* of what they actually intended to change. This has implications for programming-by-example, and there are two approaches to tackle this situation—both of which are supported in reCode.

The first approach is manual: the developer can navigate to the Summary View (Figure 2 A), hover over one of the extraneous matches (Figure 2 C), and click the 'x' icon to remove it from the search results. The second approach is to for the synthesis engine to filter matches: ReFazer* considers all search results as *candidate* additional inputs, and applies an anti-unification mechanism to discard candidates that are incompatible with the developer-provided changes. Thus, the first approach is useful if the developer wants to use the Summary View for manual investigation and bookkeeping; the second approach is useful if the developer just wants to the reduce the amount of work needed to do their code transformation.

5.1.2 Step 2: Transforming Code by Example. Through our formative study, we learned that a barrier to correctly authoring regular expressions or scripts is the need for developers to construct a complete specification upfront. In textual or modal transformation tools, the developer also typically needs to know the tool exists and learn the syntax of a language or the UI to perform their code transformation task.

By contrast, reCode lets developers demonstrate program changes *directly* in the editor. Essentially, developers are able to construct this specification incrementally through a more intuitive editing affordance. reCode's by-example workflow is also designed to solve the problem of discoverability and provide better context. Because the developer types their examples within the main editor pane (Figure 2 D), they can take advantage of the full range of editor support, including syntax highlighting and auto-completion.

[2]https://code.visualstudio.com/blogs/2017/02/12/code-lens-roundup

5.1.3 Step 3: Iteratively Refining the Synthesis Results. The interaction so far has been developer-initiated. But once the developer types their first code transformation, the synthesizer takes the initiative.

ReFazer* accepts each developer's code transformation as a positive input-output example to drive synthesis, uses the search results as candidate locations, and returns suggestions to the user interface. reCode renders these suggestions directly in the editor as Inline Diffs—the original code is highlighted in pink and the suggested replacement is highlighted in green (Figure 2 E F). Users can act on each suggestion by clicking Accept, Reject, or Accept All in This File above the suggestion. We decided to limit Accept All to the current file based on our formative study, where participants were reluctant to accept all changes from a code transformation tool without inspection (however, "Accept All" is available under the kebab menu to the lower right of Figure 2 B).

Since the synthesizer is operating in the background in a blackbox manner, the visibility of system status is an important aspect of reCode. In addition to code diffs inline, the Summary View also conveys the status of the synthesizer by directly highlighting search results: Green highlights indicate original edits done by the user; Yellow highlights indicate matches with available suggestions; Blue highlights show the current selection.

One consideration is *when* to send the developer-provided code transformation to the synthesizer: they may be typing slowly, pausing to think, or any variety of other activities that may cause the user interface to prematurely roundtrip to the synthesizer and incorrectly update the suggestions to the developer. Our unsophisticated solution to this problem was to add a short debounce—delaying sending examples to the synthesizer until the developer pauses for a few seconds—which worked reasonably well.

Another consideration is what happens when the developer edits a line that does not match the original search. For example, consider when a developer searches for a comment like `// TODO`, but makes all of their actual edits to the line below. Again, we implemented a simple approach that constructs a *window* around the search location overlaps the edit location (in the above example, the window size is ±1). This heuristic also worked reasonably well, with the caveat that ReFazer* becomes sluggish if the window-size becomes too large.

Any suggestions the developer chooses to accept becomes an additional positive input-output example. To avoid infinite recursion, once a region of code is accepted, it will not be considered again as a candidate location for synthesis. Any suggestion that a developer chooses to reject becomes a negative input-output example, or filter. Finally, accept all changes in the current file works essentially the same as accept, but sends all of the accepted suggestions at once.

These iterations continue until the user closes the find UI.

5.2 Overview of ReFazer*

ReFazer* [16] is a robust, general-purpose synthesizer that reasons about differences in abstract syntax trees to learn code transformations. Although ReFazer* is not specifically designed to support a find-and-replace user experience, the engine has several properties that reCode is able to usefully exploit. This section presents a high-level overview of these properties; detailed formal semantics of ReFazer* and its full performance benchmarks can be found in Gao et al. [16].

Gao et al. [16] report that with only one input-output example, ReFazer* can learn a correct program transformation with 96% precision (through a benchmark of 12,642 test cases ranging from single-statement to multi-line edits). With two examples, the precision increases to 98%, and with three examples, 99%. For all three cases, recall is above 99%. Because we rely on ReFazer* for program synthesis, we expect reCode to have similar performance for comparable tasks.

ReFazer* frames code transformation as a semi-supervised learning problem. In addition to the concrete edits (input-output pairs) that the technique uses as instances, the learning process also exploits access to additional inputs—that is, program subtrees—if they are provided to the synthesizer. Conveniently, this interface for ReFazer* maps closely with the user experience needs for reCode's find-and-replace: the developer's initial search results become the additional inputs, and the developer's subsequent code transformations correspond to input-output subtrees. ReFazer* applies a strategy of anti-unification that discards incompatible additional locations. From reCode's perspective, this means that we do not need special handling to support developers who apply simple keywords, resulting in a superset of the actual locations required.

ReFazer* requires the developer to enter a special mode to provide examples and feedback to the system. While this can be a limitation for modeless user interfaces, it is precisely the interaction model for find-and-replace because developers enter an explicit mode.

Because ReFazer* works on abstract syntax trees, we built a shim layer that sits between the front-end and ReFazer*; this shim takes lines of text and rewrites them them into trees and vice versa. Modern compilers offer built-in APIs to facilitate this, so our integration work more or less involves invoking the appropriate facilities.

5.3 Limitations and Future Work

User interface. When the number of search results are large, developers are likely to hit perceptual and cognitive scalability limits that make it overwhelming to make sense of and navigate the search results. One solution to push these scalability limits outwards would be to apply an intelligent clustering algorithm that groups "related" changes together, and only present one exemplar search result from that group. For instance, one type of relation might be to might cluster matches by their relative location in the program, such as "all `bind` calls in constructors." Existing research on detecting code clones [10, 20, 24] can serve as inspiration for how to group code transformations in the Summary View.

Although we implemented an inline diff within the editor, our approach was less-than-ideal due to limitations in the Visual Studio Code extension API. Specifically, Visual Studio code already provides a high-fidelity inline diff experience for comparing version control changes, but this facility is not exposed in a way that extension authors can use. Although our inline diff is conceptually similar, it lacks some of the niceties like syntax highlighting, theme support, and support for diffing long lines.

In our design, we made an assumption that developers in the find-and-replace will only make relevant edits. That is, only edits are intended to be used as part of the ReFazer* synthesis process. However, it is possible that developers might make unrelated, interleaving edits (for example, they might fix a typo while making a code transformation). A future implementation should consider options for addressing this scenario. One possibility is to allow the developer to explicitly pause the mixed-initiative loop when making an unrelated edit. Another option would to be incorporate the concept of noisy edits within the ReFazer* engine itself.

Developers may unintentionally provide *ambiguous* or *conflicting* code transformation examples. For instance, `f(a, b)` to `g(b, a)` is ambiguous if the developer provides an example `f(c, c)` to `g(c, c)` demonstrate renaming and swapping the arguments. Similarly, code transformations can also be conflicting: `a` to `b` and also `a` to `c`. For ambiguous code transformations, the developer must inspect the transformation closely. For conflicting code transformations, we surface a generic error message to the developer. However, an improvement to this user experience would be to provide an explanation for why one or more code transformations conflict.

Program synthesis engine. ReFazer* is useful for a variety of code transformation tasks, but currently has some known limitations. Because ReFazer* is tree-based, it works at the node level and does not perform substring-to-substring transformations. For example, `translate` to `tranform` works, but `translateObject` to `transformObject` would not. To support this scenario, ReFazer* could be extended by adding FlashFill-style string transformations [17].

One scenario that is not handled by ReFazer* are code transformations that require reasoning about a countable but arbitrary number of nodes in the tree. For example, consider the program:

```
new string[] {
    a.ToString(),
    b.ToString(),
    c.ToString() }
```

which the developer wants to transform to:

```
new string[] {
    a.ToString(),
    b.ToString(),
    c.ToString() }
```

The problem is that this code transformation requires generalizing to an arbitrary number of elements in the array—this is not supported in ReFazer*. The current workaround is for the developer to do repeated find-and-replace tasks for arguments of length one, length two, length three, and so on up to the largest number of arguments.

ReFazer* does not understand the concept of associated edits. For example, the Multiloc in our usability evaluation requires the developer to delete the line with bind, as well as modify the corresponding function having the same function name. To allow this, an extension to ReFazer* implements a heuristic that treats this task as two independent synthesis tasks: one for deleting bind, and another for the function modification. The consequence of this is that a developer might accidentally delete a bind and forgot to modify the corresponding function, and ReFazer* would be unable to detect this error.

ReFazer* is resilient to variations in program text (for example, whitespace, newlines, and other formatting trivia) and tries to mimic the formatting that developers do as best-effort. However, there is no guarantee that the suggested code transformation will preserve formatting in same style as the input example, and this annoys developers.

6 USABILITY EVALUATION OF RECODE

6.1 Participants and Setup

We recruited 12 participants (10 men, 2 women, mean self-reported experience of 6.8 years) using the same recruitment survey described in Section 4. Participants are denoted as P1-P12 in subsequent sections. For programming languages, participants in their day-to-day tasks report using TypeScript (4), Python (2), C# (8), C++ (4), with some reporting more than one language. On a 5-point Likert-type scale, participants reported the frequency of code transformation tasks to be: very frequently (2), frequently (5), occasionally (3), rarely (2). Participants also reported their familiarity with VSCode: extremely familiar (2), moderately familiar (8), somewhat familiar (2).

Each session took 45-60 minutes and was conducted remotely on Microsoft Teams. Developers connected to a remote desktop environment pre-configured with reCode. All sessions were audio and video recorded, including participants' screens.

6.2 Tasks

In the formative study, participants discussed the challenges they had endured when transforming code and several participants shared recent transformation tasks. Through the formative study, we designed four tasks (Table 1) that represent increasingly complicated code transformations. We identified an applicable public GitHub repository for each task. We then selected a subset of the files so that the size of each task ("Required changes" in Table 1) reflects the "murky middle" (15-50 lines-of-code changes), in which we expected the participants to make an deliberate decision on whether to use a tool or perform the task manually.

(1) Constant-string[3] replaces a constant string in an entire program. This transformation is supported by almost all editors through find-and-replace or rename refactoring. All participants in the formative study reported frequently making this kind of change.
(2) Gather-args[4] gathers arguments from chained function calls into a single call. This transformation requires more effort since the arguments from the found code needs to be reused in the replacement. This transformation might be accomplished by using regex-based find-and-replace with capture groups. Formative study participants reported that refactoring function calls is common, but also demand significant effort. For example, F2 reported "copy-pasting and editing lots of function calls in a test suite".
(3) Multiline-add[5] finds one existing line of code, changes this line, and appends additional code. The task represents changes involving a single-line match and multi-line changes, such as adding a null-pointer check around a line of code, or breaking up a line of code into multiple lines. The task requires developers to take extra care to handle formatting and newlines, and might be accomplished using a keyboard macro or a multi-line regular expression.
(4) Multiloc[6] changes two separate locations that are connected by a common method name (*e.g.*, `func` in Table 1). This transformation involves multiple matches and changes, which is common in language migration and design-pattern changes [11]. Specifying such transformations in one regex or macro is challenging since they depend heavily on syntax and formatting. Therefore, this task is often accomplished with more complex tools that manipulate programming language structure like Comby [50] or AST transformers such as `jscodeshift`.

Since we did not require our participants to have experience with a specific programming language, we provided them with a before-and-after example to illustrate the kind of code transformation they would need to perform for each task.

6.3 Protocol

To reacquaint participants with code transformation tasks, participants started by performing a warm-up exercise using VSCode without reCode. In this exercise, we asked participants to change from `t.is(a, b)` to `expect(a).toEqual(b)` (17 lines in 5 files). We then showed participants a short reCode tutorial. Afterwards, they performed the remaining tasks in random order using VSCode with reCode (Section 5). Participants were free to access online resources during all tasks.

After completing the transformation tasks, participants were given a questionnaire. The questionnaire asked them to self-evaluate the difficulty and tediousness of each task on a 5-point Likert scale (Strongly disagree–Strongly agree). To validate the relevance of the tasks, we also included a question asking how frequently participants encountered similar tasks in their work. The questionnaire also asked if the participant would use a production version of reCode. At the end of the evaluation, we conducted a retrospective interview to gather feedback about reCode.

7 RESULTS

In this section, we describe our participants' task performance, their responses to the follow-up questionnaire, and feedback from the retrospective interview.

7.1 Efficiency and Effectiveness

Table 2 shows the average time taken and number of participants that successfully completed each task. After each task, participants were asked to rate the frequency of which they encounter similar

[3]The code for Constant-string is adapted from a test file of the `svgpath` library: https://github.com/fontello/svgpath.
[4]The code participants received for Gather-args is one of the transformations required to migrate from `Jest` to AVA.js: https://jestjs.io/docs/migration-guide.
[5]The code for Multiline-add is one of the breaking changes introduced by `v9` of `next.js`. The authors of the library provided a script to automate this complex change: https://nextjs.org/docs/upgrading.
[6]Multiloc is a structural change for using a new language feature of JavaScript ES6. An implementation for this particular task can be found in `react-codemod`: https://github.com/reactjs/react-codemod.

Table 1: Tasks for the usability evaluation. The tasks reflect the range of scenarios identified in the formative interviews: from constant strings to tree transformations. Each task represents a type of code edits developers often encounter.

Task	Code to find	Replacement code	Task type	Required changes
Constant-string	`translate`	`transform`	Plain text replacement	14 lines in 2 files
Gather-args	`expect(a).toEqual(b);`	`same(a, b);`	Function refactoring	17 lines in 5 files

Multiline-add	`export default withAmp(Box)`	`export default Box`
		`export const config`
		`  = { withAmp: true }`
			API migration	28 lines in 13 files

Multiloc	`class Example {`
	`  constructor() {`
	`    this.func =`
	`      this.func.bind(this);`
	`  }`
	`  func() {}`
	`}`
		`class Example {`
		`  constructor() {}`
		`  func = () => {}`
		`}`
			AST transformation	50 lines in 10 files

Table 2: Summary results for each task. The number of participants that completed each task and the average task time are shown. After each task, they were asked to rate (1) "this task was difficult to complete;" (2) "this task was tedius;" and (3) "I encounter similar tasks in my work." The rating scale as from left-to-right was: 🟥 disagree (1), 🟧 Disagree (2), 🟨 Neither agree nor disagree (3), 🟩 Agree (4), 🟩 Strongly Agree (5). Median values precede each distribution.

			Frequency of task		Difficulty rating		Tediousness rating	
Task	**Avg. time taken**	**# completed**	Med.	Dist.	Med.	Dist.	Med.	Dist.
Constant-string	1:35	12	5		1.5		2	
Gather-args	3:09	12	4		2		2	
Multiline-add	4:34	12	4		1.5		1.5	
Multiloc	3:22	10	3		2		2	

tasks, the difficulty of the task, and the tediousness of the task. All participants were able to complete Constant-string, Gather-args, and Multiline-add using reCode. The average completion time was less than than five minutes. Finally, two participants failed to complete (P1, P2) Multiloc because of an unexpected failure in the reCode synthesizer. The most complicated task, Multiloc, also appeared least frequently in participants daily work. Most participants encounter all other tasks frequently.

7.2 Participant Feedback

We group participants' feedback using the steps from the reCode user experience: Find, Edit, and Generalize. Finally, we report participants' feedback on the overall user experience.

Find. All participants initiated the reCode experience using the "Find" feature very early on: participants either immediately started using "Find," or they poked around a few files first, made a guess about a keyword, and then used "Find" to search for that keyword.

Most participants (P1, P2, P3, P5, P7, P8, P9, P10, P11, P12) used an overly broad keyword rather than an elaborate but precise expression (P4, P6). For instance, when performing Gather-args, P4 and P6 searched for `expect\((.*)\).toEqual\((.*)\)` , whereas all other participants searched for `toEqual` initially. Participants later added punctuation around the keyword as an *ad hoc* solution to narrow down the scope (for example, `).toEqual(`), because they "usually search for something very generic and see if I need to narrow down my search later" (P9). Some participants reported that this is "what [they] would have done anyway" (P2), with or without reCode.

Some participants expressed a desire for "structural search to prevent over-matching, because `bind` can appear anywhere and what I really want to find is all function calls of `bind` " (P1, P4). However, these participants struggled to achieve this because they "don't know how [they] would say it" (P1) and resorted to adding simple punctuation around the search term because it was "the best they could do" (P1).

After performing the find, all participants (P1-P12) manually inspected more than one results before performing any changes because they "wanted to see all the possible cases to see if [they're] overmatching" (P3). When navigating through the find results, they liked the "holistic view of all the results" (P1) in the Summary View Figure 2 A.

Change. After reCode displayed the find results, all participants (P1-12) proceeded to directly edit one of the found code locations.

Direct edits helped participants make sense of the transformation and estimate "if it's easy enough to go through things manually. If it takes more than 5 minutes, [they'll] go for other tools" (P7). After editing one or more find results, participants noticed reCode's suggestions inline and noted that reCode "figured out what [they] did" (P3) and "picked up on the pattern now that [they] did it a couple of times" (P5).

Participants appreciated that direct editing is "way faster and much easier than writing regexes" (P10) and the fact that reCode "analyzes what you are doing and you don't have to write scripts by hand" (P1). But for trivial tasks like constant string replacement (Constant-string), some participants (P4, P5) were fine with using the replace box in find-and-replace: "I was equally satisfied here [directly editing using reCode], but I might fall back to regular find-and-replace since this is not a challenging task" (P4).

Some participants (P2, P4, P6, P7) requested better visibility of the system's status. For instance, in the first task, P7 asked, "Is this running? I guess I'll just keep doing thing manually" until reCode displayed the first inline diff in their editor pane. P7 wanted to "know it's there in the first place" and "know if it starts working or not." P2 needed "more confirmation in the UI that it's searching" and P6 proposed adding "an indicator that say 'suggestion in progress' in the editor pane." P4 speculated that "exposure might be key, because after getting used to it I understand the green bar is telling me if it's active."

Generalize. All participants (P1-12) understood reCode's suggestions after seeing inline diffs and inline actions (Figure 2 E F) in the same file or yellow highlights in the Summary View (Figure 2 A).

P5 thought the inline diffs were "really cool because [they] wanted to see what things were before replacement and this way [via inline diffs] [they] can verify if everything's right." P12 said the inline diff and actions were "pretty intuitive, and just like git in VSCode. I can see the diffs inline and choose to accept or not. Very familiar." P9 preferred our inline diffs to a separate window for find-and-replace; in their editors "screen real estate is important, and [a separate diff view] is too distracting."

After viewing a few of the suggestions by scrolling around and/or clicking through search results, participants felt that "it's doing the right thing" (P12) and "trusted it like [they] trust 'Rename Variable' in VSCode" (P11). When performing Multiline-add, P6 deliberately looked for "the trickiest case" and found out "it's reusing the component names correctly, now I think it works." Some participants (P4, P6, P10) directly edited the suggestion to test if reCode would update the rest of the suggestions as well, and found that "every string gets updated after I changed one of them, great!" (P6).

After participants expressed some confidence in reCode's suggestions, all of them (P1-P12) interacted with the inline actions (Figure 2 F). For instance, P3 was "comfortable accepting all after reviewing a few items" but requested an "an 'Accept All in Project' button to finish the whole thing." However, after making the same request as P3, P2 commented that "the engineer in me says be careful. I would compile and see if anything breaks. The diligent person in me says there shouldn't be this [Accept All in Project] button to allow me to do it."

As noted in Section 5.3, reCode sometimes does not preserve the exact formatting of the developer's original edit. For instance, P4 noticed an extra new line in the suggestion and said: "Boo, it added this new line. I deleted the new line character, so should you!" For the most complicated task (Multiloc), a few participants requested the ability "to link two related edits and if I click on accept changes for

raisebox-4pt `bind`, the function below should change, too" (P10). P9 mentioned the same feature because "in [their] head, these two changes are grouped together and [they] wished the tool could show [them] how they are related."

End-to-end feedback. Participants liked the overall reCode experience because it "was really fast" (P1, P2, P10), "worked naturally" (P2, P5, P6, P12), "was easy to use" (P4, P7, P9, P12), and "saved time" (P2, P3, P4, P6, P8, P9, P10, P11). P9 noted that they "spent too much time battling things like regular expressions and this will be a huge productivity multiplier." P2 appreciated how well reCode fits into their workflow because "it's basically how I would do it normally." P11 shared their experience with auto-completion tools and said, "it's always trying to give me suggestions and I don't need them most of the time and after a while I just turned it off." Instead, P11 preferred reCode's workflow because "it's more selective. Instead of listening passively and trying to come up a plan for me, it only works when I have a plan to actively change things."

When asked whether they will use a production version of reCode in the questionnaire after the study, participants responded either "Would use" (9/12) or "Probably use" (3/12). All participants asked when reCode would be shipped officially so they can start using it. They were excited to use reCode to automate a variety of their daily tasks such as "writing repetitive tests" (P2) and "refactoring my Powershell scripts" (P4). Automatic synthesis of code transformations enabled them to have ways to perform a task "when the editor doesn't have refactoring support" (P9). P4 gave it "10 out of 10" and said, "I'd use this daily. Sometimes when I get 50 matches and I just thought I'll just do it manually, but this thing is like 'do you want me to automate it?' I love it!" P5 "loved the granularity of the tool," and P9 said that because "find-and-replace is such a common thing, the ability to do this all directly [in my editor] makes this my favorite tool."

8 DISCUSSION

The results of our evaluation suggest that reCode addresses the design goals we formulated in Section 4. Participants found reCode provides a unified entry point for code transformations (D1), offers a lightweight way to transform their code (D2), and provides useful affordances to allow developers to incrementally inspect their code transformations and compare the before-and-after-results. In this section, we discuss the benefits of reCode's unified interaction, developers' expectations about code transformation explainability, and other insights about how developers might leverage reCode.

8.1 Example-driven Intent through a Lightweight, Unified Interaction

We found that developers frequently need to make code transformations, but existing tools require them to make unsatisfying trade-offs, particularly in the "murky middle." reCode removes much of this decision-making dilemma by offering a unified entry-point for their code transformation task. When using reCode, the developer does not have to consider the cost of switching out of their editing workflow or calculate the utility of automation (D1). Instead, they find and make manual edits as usual, and automatically get non-intrusive suggestions that perform the remaining edits on their behalf.

Existing code transformation tools also force them to switch out of their editing workflow to automate these edits. For example, P9 recalled that "they don't want to switch out of my editors to do find-and-replace. We really don't like distractions from our workflows."

Our participants told us that using tools like regular expressions and AST transformers required a careful planning and authoring process. Before using any transformation tool, developers have to learn their intricacies. The cost of this learning is often a significant barrier to automating code transformation. As P7 reminded us, "if you have a problem to solve with regular expressions, now you have two problems." reCode enables developers to make a variety of code transformations without needing to turn to regular expressions or another intricate code transformation language (D2).

8.2 Expectations about Explainability

Developers are careful about code transformations, especially when an automated tool is performing the changes. Our participants expressed a desire to iteratively and incrementally develop and test their code transformations. In addition, because code transformations can have many edge cases, they were wary of transforming code without directly being able to observe the changes.

In contrast to scripts that typically operate in batch across the entire project, participants preferred the ability to interactively inspect the code transformation and verify them inline through reCode. Instead of requiring developers to make all-in decisions on the code transformation, reCode iteratively generalize developers' direct edits and provides the developer with autonomy over accepting, rejecting, or modifying individual suggestions. Importantly, the mixed-initiative workflow of reCode lets developers progressively evaluate the effect of their edits through concrete examples, while balancing automation and inspection (D3).

8.3 Reusable Code Transformations

Developers often make code transformations that are highly contextual and tailored to their own projects: while these code transformations are important for this developer, it's unlikely that they would be able to find an off-the-shelf tool that already provides the transformation they need. As a result, developers mostly performed most edits manually and repeatedly. When working with reCode, some participants thought the tool could be improved by allowing them to keep a personal "history" (P1), "export" (P3), or reusable catalog of their own transformations.

Since the ReFazer* internally learns a code transformation, one possibility is for reCode to save or serialize this code transformation so that the developer may *reuse* it at a later time without having to reinitiate a find-and-replace interaction from scratch. A more ambitious representation would to provide a *readable* representation of the code [12], perhaps by presenting the developer with a close-to-source language like Comby [50], a structural find-and-replace template, or a codemod script like `jscodeshift`.

The ability to offer the developer a readable representation of the code transformation has several benefits. If the developer is able to read the synthesized program, they may be more comfortable accepting code transformations without needing to manually inspect and verify as many locations (D3). The developer may also want to use the synthesized program to *learn* how to use one of the many code transformation languages (D2). As one example, the Gather-args task can be written as the following Comby script:

```
match template: 'expect(:[a]).toEqual(:[b])'
rewrite template: 'same(:[a], :[b])'
```

For large-scale projects, developers might use reCode to synthesize a transformation from a smaller project, and then use the script to "bootstrap" (P3) a more elaborate script for code transformations in a larger project. Alternatively, an interesting possibility is that the developer may already have a script that they want to understand, apply, or refine. In this situation, instead of bootstrapping find-and-replace with search keywords, they could bootstrap the reCode experience using their script—and use reCode, just as before, to understand or refine the script through the unified reCode interaction (D1).

9 CONCLUSION

Our formative study showed that developers struggled to automate code transformations using existing tools; as result, they abandoned these tools and often ended up performing the changes manually. To address their needs, we designed reCode, an example-driven, mixed-initiative interaction that improves on their familiar find-and-replace experience. After performing a simple code search, reCode users can demonstrate their intended changes by directly editing code, and reCode automatically learns a more general code transformation to help developers complete the task. Participant feedback from our usability evaluation suggests that the reCode example-driven experience is intuitive, complements their existing workflow, and offers a unified approach to conveniently tackle a variety of common yet frustrating scenarios for code transformations. Developers in our evaluation were enthusiastic about using reCode in their own day-to-day work.

ACKNOWLEDGMENTS

We thank developers at Microsoft for their helpful insights and for participating in the interviews and studies. This material is based in part upon work supported by the National Science Foundation under Grant No. 1910264.

REFERENCES

[1] [n.d.]. codemod. https://github.com/facebook/codemod
[2] [n.d.]. ESLint. https://eslint.org/
[3] [n.d.]. jscodeshift. https://github.com/facebook/jscodeshift
[4] [n.d.]. Pylint. https://www.pylint.org/
[5] [n.d.]. ReSharper. https://www.jetbrains.com/resharper/
[6] [n.d.]. ripgrep. https://github.com/BurntSushi/ripgrep
[7] [n.d.]. Roslyn Analyzers. https://github.com/dotnet/roslyn-analyzers
[8] [n.d.]. Sublime Text. https://www.sublimetext.com/

[9] [n.d.]. Visual Studio Code. https://code.visualstudio.com/

[10] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance*. 368–377. https://doi.org/10.1109/ICSM.1998.738528

[11] Danny Dig and Ralph Johnson. 2006. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice* 18, 2 (2006), 83–107. https://doi.org/10.1002/smr.328

[12] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, 1–12. https://doi.org/10.1145/3313831.3376442

[13] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*. 422–431. https://doi.org/10.1109/ICSE.2013.6606588 ISSN: 1558-1225.

[14] Jeffrey E. F. Friedl. 2006. *Mastering Regular Expressions* (3rd ed. ed.). O'Reilly.

[15] Yuzo Fujishima. 1998. Demonstrational automation of text editing tasks involving multiple focus points and conversions. In *Proceedings of the 3rd International Conference on Intelligent User Interfaces (IUI '98)*. Association for Computing Machinery, 101–108. https://doi.org/10.1145/268389.268408

[16] Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachiappan Nagappan, and Ashish Tiwari. 2020. Feedback-driven semi-supervised synthesis of program transformations. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 219:1–219:30. https://doi.org/10.1145/3428287

[17] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. Association for Computing Machinery, 317–330. https://doi.org/10.1145/1926385.1926423

[18] Sumit Gulwani. 2016. Programming by examples. *Dependable Software Systems Engineering* 45, 137 (2016), 3–15.

[19] Eric Horvitz. 1999. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI '99)*. Association for Computing Machinery, 159–166. https://doi.org/10.1145/302979.303030

[20] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (July 2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[21] A. Ketkar, A. Mesbah, D. Mazinanian, D. Dig, and E. Aftandilian. 2019. Type migration in ultra-large-scale codebases. In *Proceedings of the 2019 International Conference on Software Engineering (ICSE '19)*. 1142–1153. https://doi.org/10.1109/ICSE.2019.00117

[22] Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. Association for Computing Machinery, 309–319. https://doi.org/10.1109/ICSE.2009.5070531

[23] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An empirical study of code clone genealogies. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '13)*. Association for Computing Machinery, 187–196. https://doi.org/10.1145/1081706.1081737

[24] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen. 2016. Automatic clustering of code changes. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 61–72.

[25] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2001. Learning repetitive text-editing procedures with SMARTedit. In *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers Inc., 209–226.

[26] Toshiyuki Masui and Ken Nakayama. 1994. Repeat and predict: Two keys to efficient text editing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '94)*. Association for Computing Machinery, 118–130. https://doi.org/10.1145/191666.191722

[27] David Maulsby and Ian H. Witten. 1997. Cima: An interactive concept learning system for end-user applications. *Applied Artificial Intelligence* 11, 7-8 (Oct. 1997), 653–671. https://doi.org/10.1080/088395197117975

[28] Lee E. McMahon. 1990. Sed: A non-interactive text editor. In *UNIX Vol. II: Research System (10th ed.)*. W. B. Saunders Company, 389–397.

[29] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Sydit: Creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11)*. Association for Computing Machinery, 440–443. https://doi.org/10.1145/2025113.2025185

[30] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. 502–511.

[31] T. Mens and T. Tourwe. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering* 30, 2 (Feb. 2004), 126–139. https://doi.org/10.1109/TSE.2004.1265817

[32] L. G. Michael, J. Donohue, J. C. Davis, D. Lee, and F. Servant. 2019. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 415–426. https://doi.org/10.1109/ASE.2019.00047

[33] Robert C. Miller and Alisa M. Marshall. 2004. Cluster-based find and replace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. Association for Computing Machinery, 57–64. https://doi.org/10.1145/985692.985700

[34] Robert C. Miller and Brad A. Myers. 2001. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USENIX Association, 161–174.

[35] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 143 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360569

[36] Maxim Mossienko. 2004. Structural search and replace: What, why, and how-to. *OnBoard Magazine* (2004).

[37] E. Murphy-Hill and A. P. Black. 2008. Refactoring tools: Fitness for purpose. *IEEE Software* 25, 5 (Sept. 2008), 38–44. https://doi.org/10.1109/MS.2008.123

[38] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 5–18. https://doi.org/10.1109/TSE.2011.41 Conference Name: IEEE Transactions on Software Engineering.

[39] B. A. Myers. 1990. Invisible programming. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*. 203–208. https://doi.org/10.1109/WVL.1990.128407

[40] Brad A. Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (May 2016), 62–69. https://doi.org/10.1145/2896587

[41] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, 180–190. https://doi.org/10.1109/ASE.2013.6693078

[42] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton. 2019. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) (ICSE '19)*. 819–830. https://doi.org/10.1109/ICSE.2019.00089

[43] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. 2010. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. Association for Computing Machinery, 315–324. https://doi.org/10.1145/1806799.1806847

[44] Robert P. Nix. 1985. Editing by example. *ACM Transactions on Programming Languages and Systems* 7, 4 (Oct. 1985), 600–621. https://doi.org/10.1145/4472.4476

[45] Andreas J. Pilavakis. 1989. The vi Editor. In *UNIX Workshop*, Andreas J. Pilavakis (Ed.). Macmillan Education UK, 59–65. https://doi.org/10.1007/978-1-349-19900-6_6

[46] Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin Vechev. 2013. Refactoring with synthesis. *Proceedings of the ACM on Programming Languages*, 339–354. https://doi.org/10.1145/2509136.2509544

[47] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 404–415. https://doi.org/10.1109/ICSE.2017.44

[48] Richard M Stallman. 1981. EMACS the extensible, customizable self-documenting display editor. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*. 147–156.

[49] Atsushi Sugiura and Yoshiyuki Koseki. 1996. Simplifying macro definition in programming by demonstration. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology (UIST '96)*. Association for Computing Machinery, 173–182. https://doi.org/10.1145/237091.237118

[50] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight multi-language syntax transformation with parser parser combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, 363–378. https://doi.org/10.1145/3314221.3314589

[51] Louis Wasserman. 2013. Scalable, example-based refactorings with refaster. In *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools (WRT '13)*. Association for Computing Machinery, 25–28. https://doi.org/10.1145/2541348.2541355

[52] Andrew J. Werth and Brad A. Myers. 1993. Tourmaline (abstract): Macrostyles by example. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93)*. Association for Computing Machinery, 532. https://doi.org/10.1145/169059.169532