Interactive Extraction of Examples from Existing Code

Andrew Head, Elena L. Glassman, Björn Hartmann, Marti A. Hearst
UC Berkeley
Berkeley, CA, USA
{andrewhead, eglassman, bjoern, hearst}@berkeley.edu

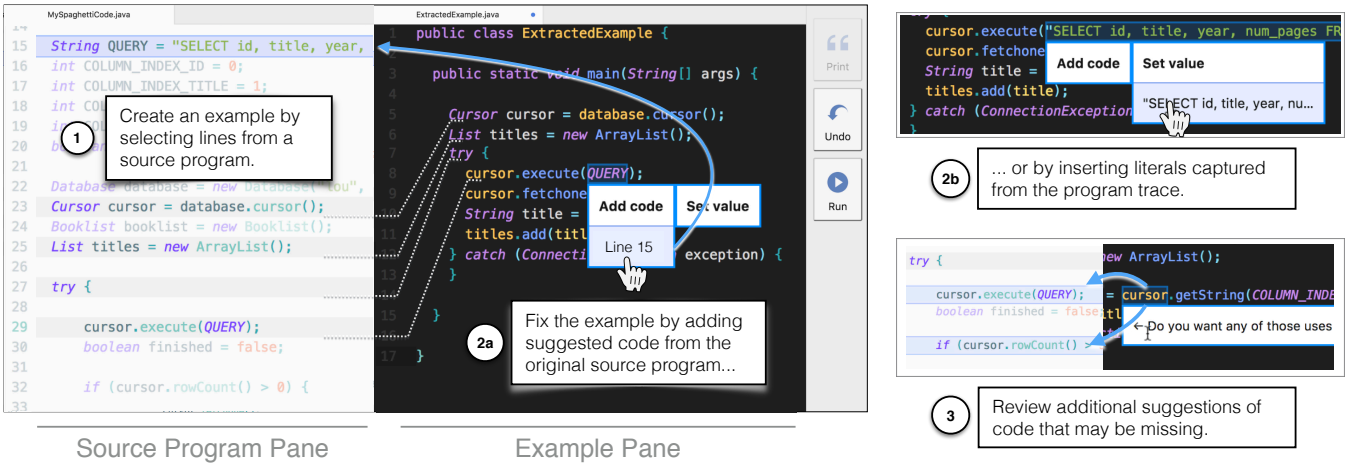


Figure 1: Extracting example code from existing code with CodeScoop. With CodeScoop, (1) a programmer selects a few lines they want to share from a source program, and CodeScoop helps them build them into a complete, compilable example. To help programmers make complete examples, CodeScoop detects errors and recommends fixes by (2a) pointing to potentially missing code and (2b) suggesting literal values from the program trace that can take the place of variables. (3) It also recommends code the programmer may have overlooked, like past variable uses and nearby control structures.

ABSTRACT

Programmers frequently learn from examples produced and shared by other programmers. However, it can be challenging and time-consuming to produce concise, working code examples. We conducted a formative study where 12 participants made examples based on their own code. This revealed a key hurdle: making meaningful simplifications without introducing errors. Based on this insight, we designed a mixed-initiative tool, CodeScoop, to help programmers extract executable, simplified code from existing code. CodeScoop enables programmers to "scoop" out a relevant subset of code. Techniques include selectively including control structures and recording an execution trace that allows authors to substitute literal values for code and variables. In a controlled study with 19 participants, CodeScoop helped programmers extract executable code examples with the intended behavior more easily than with a standard code editor.

CHI 2018 April 21–26, 2018, Montreal, QC, Canada

ACM ISBN 978-1-4503-5620-6/18/04.
DOI: https://doi.org/10.1145/3173574.3173659

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

Author Keywords

programming support; example sharing

INTRODUCTION

Code examples are a key format for knowledge exchange between programmers. Examples provide an essential resource to learn about tools, and a starting point for writing new code [28, 29]. Examples can demonstrate best practices for using particular APIs and confirm programmers' hypotheses about how things work [27]. As a result, HCI and software engineering research has focused on how to support the life cycle of working with examples, e.g., authoring multi-stage code examples [10, 14], searching for examples [6, 12, 31], and integrating examples into one's own code [22, 38].

Yet examples are often missing or insufficient for many programming tasks. Around 14% of how-to questions on popular Q&A platforms may not receive answers [33]. Even for APIs that appear well-documented in online Q&A platforms, high coverage can take years to achieve and miss important topics [23]. Even if examples are available, they may not be self-explanatory: many lack important code required to run

or understand them [34]. While programmers can find usage examples in existing code, like unit tests, looking for examples in code takes time, and borrowing from incomplete examples in documentation can be error-prone [18].

In this paper, we aim to improve the state of the art in code example production. Good code examples are concise [19, 20, 28] and focused [19, 28]. Additionally, executable examples allow programmers to re-run and re-mix them, i.e., learn from, experiment with, and modify them for their own purposes.

A programmer's own code project can be a source of good examples. However, when the programmer attempts to extract an example from one of their projects, it may be time-consuming to separate out extraneous dependencies and logic that are unrelated to the concise and focused example they envision [24]. To understand this authoring task better, we ran a formative study in which 12 programmers each authored a code example based on code they had previously written.

Observations from this formative study led to the design of CodeScoop, a tool that helps a programmer make *scoops*–or focused, executable examples—from existing code. We offer the idea of a scoop as a refinement to program slices. With program slicers [32], programmers point to specific lines of code, and a slicer extracts a subset of lines required for those lines to run correctly. With a code *scooper*, a programmer and a tool work together in a mixed-initiative dialogue to extract, simplify and clarify code. A slice is finished when code has been extracted that computes the same result as the full program. In contrast, a scoop is finished when the code has the intended behavior, which could be different from the original program, and is concise and readable.

To scoop code, a user selects an initial set of lines from a source program. The user and tool work together to iteratively add important code. CodeScoop flags errors, suggests potentially relevant code, and offers fixes derived from static and dynamic analysis of the source program (Figure 1). Scoops can be tested like ordinary code, by compiling and running them with the editor. Example code produced with CodeScoop can then be shared in many of the ways already used by programmers: it can be integrated into tutorials, answers on Q&A sites, or published in a public repository.

We conducted a controlled study to gain insight into how CodeScoop could support example extraction from existing code, versus comparison tools. Participants were successful at extracting example code: using CodeScoop, 16 out of 19 participants successfully extracted a code example in ten minutes or less (compared to 11 out of 19 with a standard text editor). Participants created examples by making a median of 2 selections, replying to a median of 5 suggestions, resolving a median of 2 errors with CodeScoop's help, and accepting a median of 12 automatic corrections.

Compared to text editors, CodeScoop was more enjoyable and less difficult to use; however, participants found constraints on the ability to make arbitrary direct edits to be both helpful and restrictive. Compared to program slices, scoops were often shorter and made program results more visible, though sometimes participants omitted relevant content that a slice would have included. CodeScoop also permitted multiple approaches to building a correct code example, which reflects varying viewpoints about what belongs in a code example. In summary, the contributions include:

- A mixed-initiative interaction technique for helping programmers "scoop" concise, focused, executable code examples from existing code projects,
- a proof-of-concept system which illustrates this technique for a subset of the Java language, and
- a controlled study that provides insight into how CodeScoop supports the extraction of examples from existing code.

RELATED WORK

Tools for Authoring Example Code

Some tools help programmers *author* examples and tutorials. Ginosar et al. create an IDE extension that helps authors create multi-stage code examples by propagating changes (insertions, deletions and modifications) to multiple saved versions of their code [10]. JTourBus offers a tool for creating tour-structured documentation of large amounts of Java code [21]. JTutor is a software suite consisting of two Eclipse plug-ins that help instructors package tutorials based on Java code and enable students to replay these tutorials [14]. In contrast to these tools, our tool aims to help programmers create executable example code from their existing code.

Tools for Example-Centric Programming

More broadly, HCI and software engineering research has recently focused on how to support the life cycle of working with examples, including search engines for locating examples [6, 12, 31] and understanding alternatives within large corpuses of examples [11], and development tools for integrating examples into one's code [6, 22, 38]. The Bing developer assistant [39] is an editor extension that proactively recommends similar examples. Our tools contributes to this ecosystem of tools to support example-centric programming, providing a new interaction technique to aide in the production of executable code examples from programmers' existing code.

Program Slicing

In the domains of program comprehension and software engineering, program slicing techniques aim to help programmers separate code that represents a single concern from a large, tangled code base (for a review, see [32]). Program slicing has been offered as a technique for helping programmers understand code and make derivative, simpler programs from existing code, and has been included in both commercial and experimental tools including Frama-C [9], CodeSurfer [3], and Indus [13]. However, there is limited existing research with human subjects that evaluates the usability of such tools for the purpose of deriving simpler programs or supporting program exploration (one exception being the program slicing visualization techniques by Krinke et al. [15]). Our tool adapts the techniques of program slicing by suggesting resolutions to undefined variables by tracing dataflow dependencies through the example author's original code. Our work uniquely focuses on supporting a mixed-initiative dialogue between an example

author and an editor as they search for a concise, executable reduction of the original code.

Automatic Generation of Code Examples

Recent publications have shown that API usage examples can be automatically mined or synthesized from existing code, with little human input [7, 16, 17, 40]. These algorithms demonstrate the production of both "abstract" [7, 16] examples that document an important programming pattern in readable, yet uncompilable snippets, and "concrete" [17] code examples, which can be compiled and run. SpyREST [30] generates REST API examples from an API server's request and response logs. Pradel et al. propose an approach to mining patterns from existing code, using dynamic analysis to build FSMs of call sequences [26]. These techniques do not, however, allow an example author to shape or influence the result, as they might reasonably wish to, e.g., in order to answer a particular question or tailor an example for a specific audience.

Intelligent Coding Assistance in IDEs

For years, integrated development environments have provided tools for programmers to apply "quick fixes", or automatic resolutions that repair their code to overcome compiler errors, and "quick assists", suggestions for optional refactorings that could clean or improve the quality of their source code. Programmers use such tools to refactor their code [35], and recent research has provided new quick fixes (e.g., [36]) and redesigned quick fixes to encourage programmers to explore the design space of error resolutions [4]. Quick fixes and quick assists provide us with a motivating example for a familiar interaction paradigm with which to structure a mixed initiative dialogue between a programmer and a code editor. CodeScoop adds breadth to the design space of quick fixes with resolutions that integrate the runtime behavior of variables and objects, and that address resolutions relevant to authors creating both concise and executable example code.

FORMATIVE STUDY

We conducted a formative study to understand the process that programmers follow when creating executable code examples from their own code, and the obstacles they encounter along the way. We observed 12 programmers as they created example code. Participants were recruited from our professional networks, local MeetUps, and computer science researchers from a local university.

This study and a review of literature on code examples led to design recommendations for improving the user experience of extracting code examples from existing code (Figure 2). We refer the reader to Section A1 of the auxiliary material for protocol details and observations from the formative study.

THE CODESCOOP USER EXPERIENCE

The CodeScoop user interface aims to improve the example authoring process with two unique affordances. First, it helps authors replace lines of code that could contain distracting complexity with meaningful literal values. Second, it infers and recommends code inclusions that could enhance an example's adherence to the author's intent, like missing control structures and variable modifications. Here, we illustrate the user experience of CodeScoop with an example walkthrough. We refer the reader to the video figure and artifact[1] accompanying this paper to see the full "scooping" process.

Authors made examples by...	**Tools should help authors...**
Copying the original code and pasting into example editor	• Create examples from text selections • Add lines from original code at any time
Replacing variables with meaningful literal values	• Review and insert literal values that preserve program behavior
Tweaking comments and code format for readability	• Directly edit code to add comments, group lines, and add `print` statements
Making examples could be time-consuming because...	**Better tools could...**
Authors left out code	• Suggest lines of code that the current example needs to run • Add missing code automatically when it's the only sensible fix
Authors introduced errors via transcription or edits	• Constrain manual code edits • Enable early and frequent testing
It took time to remove irrelevant code	• Start from a blank file • Omit code except for explicit code selections and necessary fixes

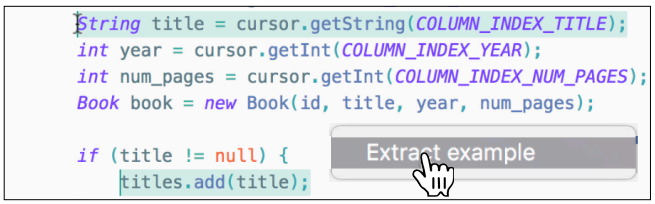
Figure 2: Tool recommendations for improving example extraction. From a twelve-participant formative study, we found that making code examples from existing code can be tedious and error-prone. Tools could help programmers extract examples by constraining manual edits, and by proposing fixes based on the source program.

Problem Setting

Lou is a programmer working on a web application. She needs to write code to query records from a database using an archaic, poorly-documented API. After reading source code, inspecting the runtime state of the database objects, and overcoming numerous compiler errors, she develops the code that she needs. The code is non-trivial, involving a query to a database, iteration over query results, and catching connection exceptions. After taking so long to figure out how to write the code, Lou wants to make the task easier for others by writing a short executable example that others can read and run.

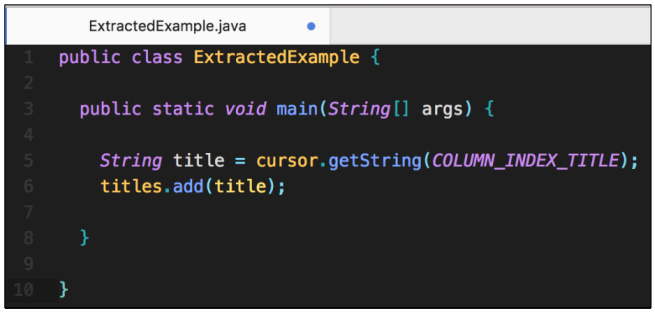
First Step: Initial Text Selections

Instead of creating an executable code example from scratch, Lou uses *CodeScoop*. From her code editor, she selects lines from her program that must be in the code example, which retrieve the data from the database cursor and save it to a data structure. She right-clicks on the selected code and chooses *Extract example* from a drop-down menu.



[1] See the project web page at https://codescoop.berkeley.edu

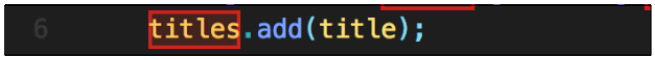
The editor splits into two panes. The left pane shows the unchanged source program , and the right pane shows the work-in-progress code example (see Figure 1). At first, the code example only contains the text selections wrapped in the `main` function of a class that can eventually be compiled and executed. The example is currently far from complete:



Mixed-Initiative Dialogue: Completing the Example

The CodeScoop editor starts a mixed-initiative dialogue with Lou to interactively make the new code example both concise and executable using the complete source program code and execution trace. The execution trace is captured from the most recent execution of the application.

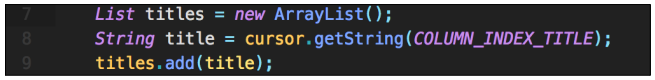
Defining Variables by Adding Missing Code: CodeScoop detects all undefined variables in the current example. Via highlighting, it indicates variables that need to be defined by highlighting the offending variable uses in red.



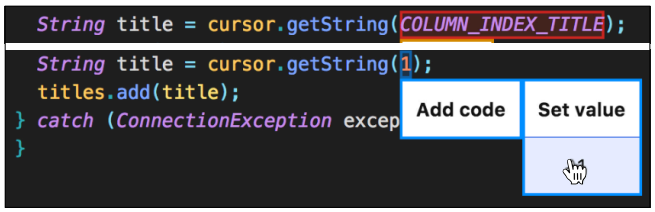
Lou hovers over the undefined `titles` variable and clicks on the "Define" button that appears below it. CodeScoop displays a menu composed of multiple methods for defining this variable (for examples, see Figure 1.2a and Figure 1.2c).

In this case, the best option is to just add the line of code that defined `titles` in the original program. Lou hovers over the line number for the suggested definition ("Line 25"). CodeScoop highlights line 25 in the source program editor. If the line is currently out of view, the source program editor scrolls until the definition is within view.

Lou inspects the original code, verifies that she wants to include this definition, and clicks to include the line. CodeScoop saves Lou's choice, and immediately updates the example with the line that provides the missing definition.

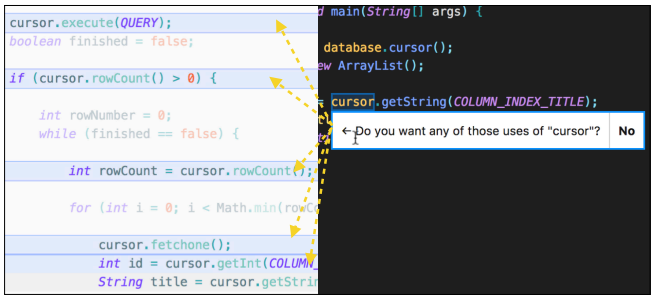


Defining Variables by Replacing Them with Literal Values: The second option for defining an undefined variable is to insert a literal value from the source program's execution trace. Lou hovers over the options for defining `COLUMN_INDEX_TITLE`: a sub-menu lists values the variable took on when the source program last ran. Here, it's just one number—1—the column index for the `title` field. Lou chooses this option, as it is more concise.



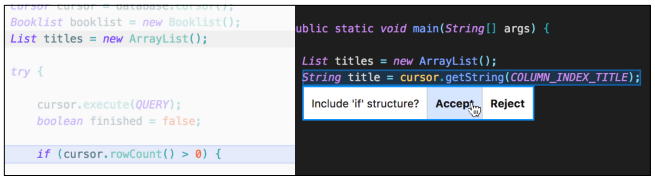
Checking for Omissions by Reviewing Previous Variable Uses: Even if all of the variables in a program are defined, this doesn't mean that the program is correct—it might be missing important modifications on already-initialized objects. To make sure Lou isn't leaving out anything important, CodeScoop points out all previous uses of variables between a variable use and the definition Lou has included.

In this case, Lou has added a line that uses the database cursor (`String title = cursor.getString(...)`) and a line that defines the cursor (`Cursor cursor = ...`). CodeScoop discovers all previous uses of `cursor`, and highlights them all for Lou's review. Lou scans over the highlighted lines...



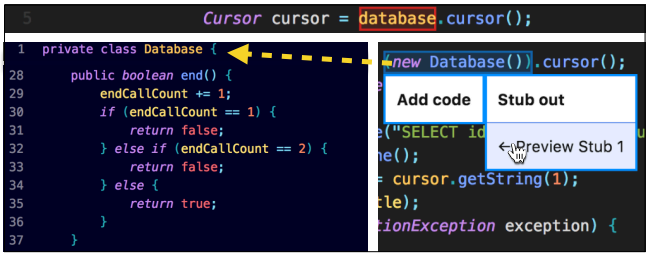
She realizes that her code example is missing two important lines that modify the state of `cursor`: `cursor.execute(QUERY)` and `cursor.fetchone()`. She clicks on the line numbers in the left gutter for these two lines, and the lines are immediately added to the example.

Including Important Control Logic and Skipping the Rest: When Lou adds code both inside and outside of an `if` block, CodeScoop asks her if she wants to include the `if` structure.



The `if` statement checks for a non-zero number of lines, but Lou thinks this check is too verbose, so she rejects it. CodeScoop also highlights a `try-catch` that handles an important exception. Lou accepts this, as she wants to show how to handle the exception in the example.

Definining Complex Data Types with "Stubs": Few people who reference Lou's example code will have access to her database's data, but they will still want a runnable example that shows how to make the call to the database and process the results. Lou decides to "stub out" the database. CodeScoop lets her resolve the database object by either including the line with the instantiation, or by inserting a stub for the database object. Lou chooses to replace the database instantiation with a stub. CodeScoop generates a new class that returns exactly the same values in the same call sequence as those from the program's execution, making it possible for anyone to run the code example without access to her database.



In this case, the stub looks too complex for Lou. She undoes the stub insertion, and chooses to define the database by including a line of code instead.

CodeScoop Fixes Trivial Bugs Automatically: Sometimes there is only one way to fix up a problem with the example code: for instance, a class has not been defined and there is only one relevant `import` statement in the original code, or a variable is missing a definition but the line has no literal values from the execution trace. In circumstances like these, CodeScoop makes the corrections automatically for Lou so she can concentrate on more cognitively demanding decisions.

Verifying Intended Behavior by Running the Code: When the code looks complete, Lou presses the "Run" button (see the right panel of the editor in Figure 1). CodeScoop compiles and runs the program, displaying the output in a bottom panel:

```
Java - ExtractedExample.java:26 ✓
[Lord of the Flies]
```

In this way, Lou verifies that all decisions up to this point have preserved the intended behavior.

Writing Annotations to Improve Readability: At this point, the code is complete, compilable, and executable. CodeScoop "unlocks" the example, now allowing Lou to make direct edits on it. Lou adds comments next to all variables and literals where a future user will likely have to provide their own data. Lou also adds some empty lines to call out which lines belong together conceptually. After she verifies again that the code compiles and runs, she copies and pastes the thirty-line executable example into an email and sends it to her coworker.

SUPPORTING THE "SCOOPING" INTERACTION

This section outlines key aspects of CodeScoop's design that follow from our formative study.

Anchor Interaction in the Code Example Itself: In the formative study, participants split attention and interaction between a code editor and the text buffer in which they wrote example code. Our tool was built to enable authors to focus primarily on the code example. Errors and suggestions are overlaid directly on the example code.

Ground Resolutions in the Original Code: Still, authors need to refer to the original code to recall the context in which the selected snippets were initially run. When they do, it should be effortless to recall that context. CodeScoop's suggestions, when referring to specific code lines or structures, call out the suggested code's context in the source program by highlighting and scrolling the source program's editor.

Prioritize Resolutions to Core Logic First: Reasoning about code can be cognitively demanding. For this reason, CodeScoop prioritizes one type of error—resolving missing variable definitions—before others. Resolving definitions leads authors through the program logic before interrupting them to fix one-off errors like missing imports and declarations.

Find the Right Time to Suggest Optional Inclusions: Some lines of code, while not necessary for compilation, may be necessary to correctly demonstrate a usage pattern. CodeScoop suggests three types of "extensions" based on an author's recent selections: (i) control structures (`if` and `try-catch` blocks, and loops) when an author selects code outside of a block after having selected code within that block; (ii) previous uses of a variable after the author has added both a use and a definition of a variable; (iii) exceptions to throw when the author adds an error-prone function call.

SYSTEM IMPLEMENTATION

Code Extraction with a Flag-Suggest-Resolve Workflow

The "scooping" process begins with a user providing a handful of text selections of what belongs in an example. From this, CodeScoop starts building the *scoop*. Internally, a scoop is represented as a set of pointers to lines that have been included from the source program, and a set of choices the user has made about what to include in the code or not.

At a high level, CodeScoop interacts with a user by following a Flag-Suggest-Resolve workflow (Figure 3). It *flags* errors and opportunities to include code when it detects important changes to the scoop. Then, it *suggests* resolutions by presenting them in a dialogue to the user. Finally, it *resolves* any problems by applying fixes to the scoop.

To flag errors, suggest resolutions, and apply resolutions, CodeScoop is built from a set of modules that analyze the program source and its execution trace (Figure 4). We describe a subset of these modules here, and refer readers to Section A2 of the auxiliary material for details about the remaining modules.

Detecting errors and relevant code

CodeScoop figures out when to prompt a user by running a suite of "detectors" on the scoop after every decision a user makes. Such detectors detect several events:

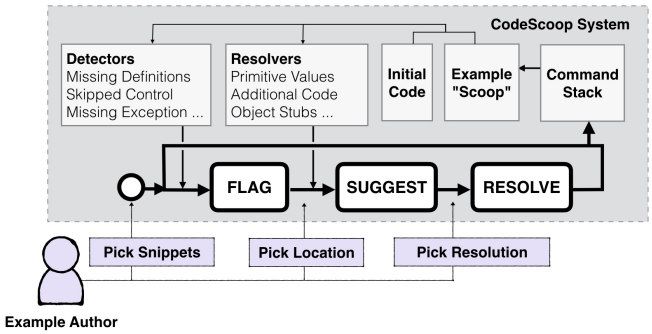


Figure 3: Iterative correction of incorrect example code. The CodeScoop system *flags* opportunities to complete and expand the code, *suggests* lists of valid resolutions for completing the code, and *resolves* completions by modifying an abstract example model called the *scoop*.

Missing definitions of variables and types

CodeScoop runs dataflow analysis to locate the character offsets of all definitions and uses of all variables in the source program, using the Soot [37] `SimpleDefUseAnalysis` program. Whenever the scoop updates with a new text selection, CodeScoop makes a list of the variables used in the scoop. It then scans all text selections in the scoop for definitions of each variable. If multiple variables are missing definitions, it highlights all undefined variables as a batch.

CodeScoop also runs an ANTLR-generated [25] parse tree walker to find all uses of types in the source program. When the user adds a text selection that uses a type (e.g., an object declaration), CodeScoop scans the scoop for `import` statements and internal classes that define the type, and flags an error if no definition has been found.

Potentially relevant control structures

An ANTLR parser is run on the source program to find all control structures. When a user adds a text selection both inside and outside of a control structure without including that control structure, CodeScoop asks if the user wants to include that control structure in the scoop.

Suggesting fixes and code additions

Whenever a user chooses an error to fix, CodeScoop runs "resolvers" to make a list of potential fixes. Fixes can come from either the source program's code, or its execution trace (Figure 4). If the resolvers only find one potential fix, CodeScoop applies the fix automatically.

Suggesting code that defines a variable

CodeScoop scans through all definitions of a variable within the same scope as the variable that is missing a definition. It recommends the line numbers of all such definitions.

Suggesting literal values for undefined variables

When CodeScoop launches, it executes the source program in a debugger virtual machine, using the Java Debug Interface [1]. As it steps through the code, it builds a table that maps a file name, variable name, and line number to a list of values each variable holds on each line of each file. When a variable is undefined in the scoop, CodeScoop looks for literal values this variable took on this line, and proposes all such values. This feature works for numbers, booleans, characters, and strings.

Fixing missing types with `import`*s and internal classes*

We use the Java Reflections API to determine the package of all of the types used in the source program. When recommending an `import` statement for the type, CodeScoop scans the imports from the source program for one that matches the fully-qualified name of the type, recommending all imports that could have provided the type. It also recommends the full text of any internal classes that define the type, as found through an ANTLR parse tree traversal.

Applying Fixes to the Scoop

After the user makes their choice, the system enters the *resolve* state, converting any fixes to transformations that can be applied to the scoop. All transformations are added to a command stack, so users can reverse any fix.

Implementation Details and Technical Limitations

CodeScoop was implemented as a plugin for the GitHub Atom code editor. It currently supports example extraction from Java programs. The plugin was written in 4,200 lines of CoffeeScript, and 1,400 lines of Java code. While IntelliJ/Eclipse have many more features for Java development, we chose Atom because it is easy to modify for prototyping novel editor interactions. Now that we have developed the interaction paradigm, it could be reimplemented in other IDEs. Supporting a new language requires a parser, a def-use analyzer, variable tracing, and reflection for methods and classes. Such tools are readily available for other statically typed languages like C#. For dynamic languages like Python, additional analysis will have to be written, (e.g., leveraging bytecode analysis [8]) to perform def-use analysis for dynamic properties.

Currently, dataflow analysis is only enabled for Java 1.4 code. The other analyses can run on Java 1.6 code and later. These limitations are specific to the libraries we chose, and not fundamental limitations for example extraction tools.

EVALUATING CODESCOOP'S DESIGN

We designed a study to gain insight into how CodeScoop or similar tools can support example extraction from existing code. We were interested in four questions:

Can programmers extract examples with the intended behavior using CodeScoop? CodeScoop incorporates a new pattern of interaction for extracting example code from existing code. Is it usable? Did it yield working examples?

How does CodeScoop compare to a standard code editor for extracting example code? In the formative study, we observed that many participants just opened an empty text editor when creating example code. Compared to this baseline, what do programmers report are the advantages and disadvantages of a tool like CodeScoop?

Could scooping decisions be automated or do they fundamentally require explicit user choice? When CodeScoop suggests code to include or literals to insert, do all programmers make the same choice? If programmers sometimes agree, maybe CodeScoop should make such decisions automatically.

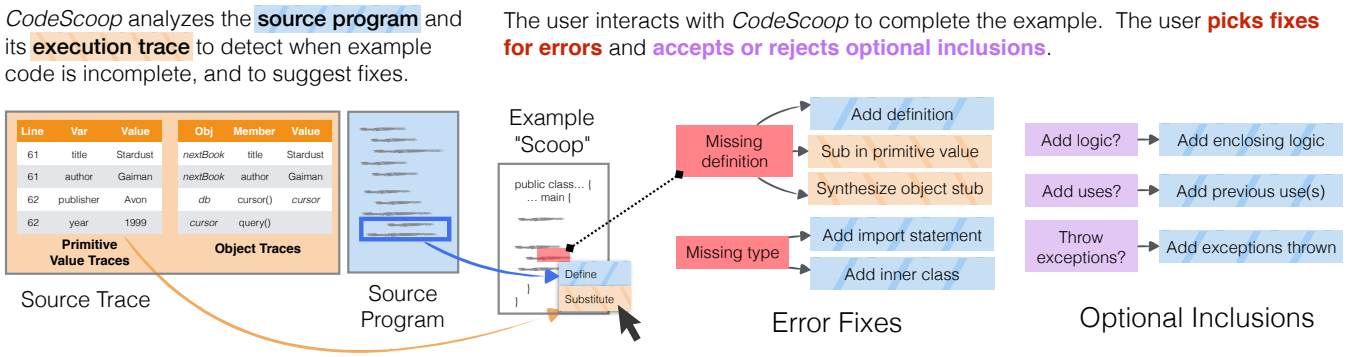


Figure 4: Suggesting fixes and code that complete a "scoop." Given a set of text selections, CodeScoop detects what's missing from an example. Whenever it detects an error with multiple resolutions, it prompts the user with a dialogue. A collection of static and dynamic analysis modules enable CodeScoop to detect missing definitions and types, propose fixes to errors, and to suggest optional code that might belong in a scoop.

Do "scoops" offer value over program slices? There is a rich literature on program slicing techniques that, given a line in source code, extract the subset of the program that affects it [32]. Do "scoops" offer advantages over such slices? Or do they give work to humans that could be done by an algorithm?

The study comprised four main tasks:

Example extraction: Participants created one example with CodeScoop and another with a standard text editor (GitHub Atom). The text editor included syntax highlighting, a button to compile and run the code, and a command to wrap a participant's initial selection in a class and `main` declaration. The editor did not have any error-checking or code-fixing functionality. While this made it more representative of the text editors many participants used to create example code in the formative study used, we note that for some participants, a development environment with error-checking and code fixes may have provided a more natural and suitable baseline.

For each example extraction task, participants were shown one of three fabricated "Stack Overflow questions", describing something a programmer might want to do with Java. Participants were asked to make an example that answered this question. Specifically, the three tasks were to:

- *Task 1*: Fetch a row from a database
- *Task 2*: Scrape text from HTML elements of a certain type
- *Task 3*: Send an email over SMTP.

Participants were also given a program from which they should extract an example that answered the question. Task 1 could be answered with an example extracted from a program 94 lines long; Tasks 2 and 3 could be answered with an example extracted from different sections of a program 135 lines long. Detailed task instructions and source programs are included in Sections A3 and A5 of the auxiliary material.

Participants were given 5 minutes to familiarize themselves with code, and 10 minutes to extract an example.

After each task, participants reported how satisfied they were with the example they made, how difficult they found example extraction with that tool, and how useful they thought the example would be for other programmers. They also rated the usefulness of each type of suggestion CodeScoop made.

Annotation interview: After a participant made their scoop with CodeScoop, we asked them to describe what code clean-ups and annotations they would make before they would feel comfortable posting it as an answer to Stack Overflow.

Follow-up questionnaire: After the two extraction tasks, participants completed a questionnaire that asked them to compare their experience with CodeScoop and the text editor.

Slice comparison: Finally, participants were asked to compare the scoop they made with CodeScoop to a slice that could have been extracted by a program slicing tool. We asked them which one would be more useful to someone looking for an answer on Stack Overflow and why.

Participants

We posted a study announcement to the social media page of the UC Berkeley Computer Science department. We enrolled 19 participants, 7 of whom were female, and 16 of whom were undergraduate students. Participants had a median of 3 years of programming experience, and 2 years of experience programming with Java (all had at least some experience). The order of tools and questions was counterbalanced to reduce any confounds due to ordering effects. Each question was answered roughly the same number of times with each tool (between 5 and 8 times, for each tool-question pair).

CODESCOOP STUDY RESULTS

Successfully extracting examples with CodeScoop

Out of 19 participants, 16 (84%) finished creating an executable example in under ten minutes when using CodeScoop. Only 3 participants did not finish creating an example in the time allotted; of these, 2 were close but encountered bugs that prevented them from finishing.

Many participants reported it was not difficult to consider CodeScoop's suggestions (Figure 5). Difficulty varied based on the type of suggestion. Deciding whether to throw an exception was not difficult—if the exception wasn't handled, the code just wouldn't compile (P13). It was trickier to make decisions about whether to include other uses of a variable,

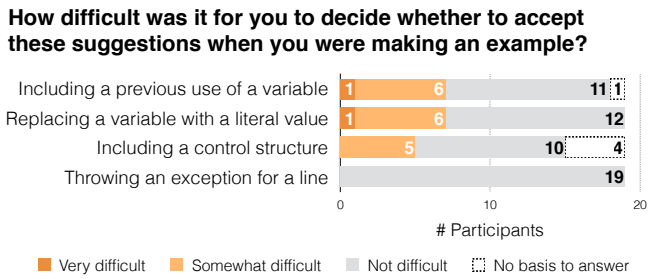


Figure 5: Not all choices in code extraction are easy. While some of CodeScoop's suggestions weren't difficult to consider (throwing exceptions for exception-prone lines of code), others required some thought (e.g., considering whether to include a previous use of a variable).

control structures, or literal values. Deciding whether to replace variable uses with values could be challenging as it required a programmer to think critically about what code really belonged (P11). Including a value could go against coding best practices of naming all the values used in the code (P13).

Scoops included a median of 1 manually-selected line ($\sigma = 5.3$) after the first selection. CodeScoop made a median of 12 automatic corrections on an author's behalf ($\sigma = 4.6$). Most of these were import statements (median = 8, $\sigma = 3.3$), though CodeScoop also fixed undefined variables by adding code (median = 3, $\sigma = 3.2$). CodeScoop automatically added one missing declaration for three participants, and inserted literal values for five participants completing task 1.

Comparing CodeScoop to a standard text editor

When asked to compare a text editor to CodeScoop, one participant aptly described the trade-offs:

> I had more freedom, but it came with a lot of pain (P14)

Participants finished extracting examples more often when they used CodeScoop than with the text editor: with the text editor, 8 of 19 participants (42%) did not finish, compared to 3 of 19 (16%) for CodeScoop (though the effect is not statistically significant using Fisher's exact test).

The median time to extract an example with CodeScoop was 5.8 minutes (σ = 1.96), and 9.5 minutes with the baseline text editor (σ = 1.52), including participants who were cut off at the 10-minute time limit. Overall, participants finished extracting examples more quickly with CodeScoop than with the baseline ($W = 76.5, p < 0.001$, Wilcoxon signed rank test). Participants who successfully extracted an example in both conditions spent an average of 2.8 minutes less (σ = 2.64) with CodeScoop than with the text editor.

On a 7-point Likert scale, extracting an example was easier with CodeScoop than with the baseline text editor (Δ (median difference) $= 3$, $W = 2$, $p < 0.01$, Wilcoxon signed rank test). It was also more enjoyable ($\Delta = 3$, $W = 4$, $p < 0.01$). Participants were more satisfied with the example they made ($\Delta = 2$, $W = 8$, $p < 0.01$), and reported the scoop would be more useful to someone learning to use the API ($\Delta = 2$, $W = 3$, $p < 0.01$). All but one participant would prefer to use CodeScoop for creating code examples in the future.

When asked to describe the advantages of the text editor over CodeScoop, 15 out of 19 participants pointed out that CodeScoop was missing the ability to make direct additions, edits, and deletions to the scoop. We do note that it would be trivial to enable direct edits for adding comments and white space.

Many participants encountered what P14 described as "pain" using a text editor. They forgot to handle or throw exceptions (P13, P15), import classes (P4, P5, P13, P14, P15, P16), and declare or define variables (P12, P13, P14, P16). They introduced errors when they moved or wrote code, like adding or removing curly braces (P12, P14, P15), or defining strings with single quotes (P12). These errors did not occur with CodeScoop: the tool handles these operations automatically. While an IDE without knowledge of a source program could help fix many of these errors, programmers may have to decide between many irrelevant options for resolving errors.

We observed one potential hazard of example extraction with CodeScoop: going on "auto-pilot" (P11), or accepting corrections without critically considering them. One participant told us, "I didn't really need to comprehend what was going on at each step–I just clicked "accept" for suggestions, with the idea that once [CodeScoop] was done, I'll manually tweak it if I need to" (P8). While reducing unnecessary program comprehension is desirable, it should not be too easy for programmers to rush through decisions that could introduce errors into their example code. This is a general tension with many tools that make corrections on a programmer's behalf.

Among the finished examples, those created with CodeScoop were shorter than those created with the text editor for task 2 (median = 22.5 vs. 34.5 lines) and task 3 (36 vs. 44 lines). Examples were about the same length for task 1 (21 vs. 20 lines). None of these differences are statistically significant.

CodeScoop allows different views of a "correct" example

CodeScoop enabled participants to choose what belonged in an example. Participants made contrasting decisions about what to include, based on differing opinions about what made a usable, readable example (see Figure 6 for one case, and the examples in Section A5 of the auxiliary material for all other examples extracted with CodeScoop).

One case where authoring decisions diverged was in the choice of whether to substitute variables with literals. For some, including literals removed unnecessary logic from the example code (P3, P11, P12, P14). For others, variable names conveyed important semantics (P1, P5), or hid otherwise private information like passwords (P2).

For almost all variables, whenever more than two participants had a choice to replace a variable with a literal, at least one person chose to define the variable with the original code, and at least one chose to replace it with a literal value; there was almost never complete agreement (Figure 7). For this study, it seems there is no "silver bullet" algorithm that could replicate every participant's extraction choices.

When CodeScoop asked a participant if they wanted to throw an exception for a line, they always accepted the suggestion. For other choices, participants' decisions were mixed: in task

```
String QUERY = "SELECT id...";
Database database = new Database(...);
Cursor cursor = database.cursor();
try {
  cursor.execute(QUERY);
  if (cursor.rowCount() > 0) {
    int rowCount = cursor.rowCount();
    cursor.fetchone();
  }
} catch (ConnectionException exception) {
}
```

(a) **Scoop 1**: Has `try-catch block`, and checks `rowCount`.

```
int COLUMN_INDEX_ID = 0;
int COLUMN_INDEX_TITLE = 1;
int COLUMN_INDEX_YEAR = 2;
int COLUMN_INDEX_NUM_PAGES = 3;
Database database = new Database(...);
Cursor cursor = database.cursor();
cursor.execute("SELECT id...");
cursor.fetchone();
int id = cursor.getInt(COLUMN_INDEX_ID);
String title = cursor.getString(COLUMN_INDEX_TITLE);
int year = cursor.getInt(COLUMN_INDEX_YEAR);
int num_pages = cursor.getInt(COLUMN_INDEX_NUM_PAGES);
Book book = new Book(id, title, year, num_pages);
System.out.println(title);
```

(b) **Scoop 2**: Wraps row in `Book`, defines column variables.

Figure 6: There's more than one way to scoop code. Participants didn't always agree on what belonged in an example. Here are two solutions to fetching a row from a database, created by two participants working with CodeScoop.

1, participants rejected control structures a median of 3.5 times, and accepted them a median of 1 time. There was no single control structure all participants either accepted or rejected.

Comparing scoops to program slices

Of the twelve[2] participants who compared their code to the slices, 9 preferred the scoop they made, and 3 preferred the slice. Participants often found that the scoops were more concise than the slices. After a first look, one participant laughed and told us that the slice "already looked gross" (P11). A lot of the slice's content didn't appear relevant:

> "I feel like there's a lot of code that isn't required to get the use of the library, and it goes through it... creates the whole HTML message, it pulls everything from Craigslist, it's all this unnecessary stuff..." (P2).

Scoops also exposed results from the example that slices sometimes missed. Several participants pointed out that their scoop collected results in a list or displayed a result with a `println` statement (P4, P7, P16), while the slice did not.

Sometimes, the slice was more concise than the scoop. One participant second-guessed their choices when they saw a slice leave out code that they included (P17). Another participant suggested the slice was more concise because some of their initial selections in CodeScoop were difficult to reverse (P16).

[2]For this comparison, we exclude 7 of 19 participants because for one task, the slice we created was incorrect. We retain the participants' qualitative data but do not report their preference.

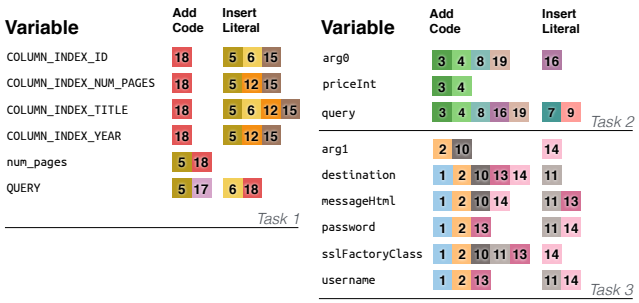


Figure 7: Choices about resolving undefined variables are, well... variable. When deciding whether define variables with additional code or a literal value, some participants replaced them with literals, and some defined it by adding lines from the original code. Each numbered box in the diagram above represents a participant resolving a variable with either code or a literal value, where decisions were often split. Algorithms will require a nuanced understanding of author preference and code semantics in order to replicate these differences.

Slices sometimes contained code that participants decided they wanted in their scoop after seeing the slice. For task 1, 3 participants preferred the more realistic API use case in the slice (P5, P17, P18). P5 appreciated seeing other relevant API calls for iterating over a database. P17 and P18 realized a reader may want more context than that provided by an example that fetched only "one row" from a database.

Slices could seem more trustworthy than scoops. One participant qualified their preference for the scoop with, "If mine works, then... I'm not sure it does, but if it did..." (P13). Others questioned whether inserting literals would break the program's behavior (P9), and believed the slice would handle edge cases their scoop would not (P1). We note however that two of these participants (P1, P13) had encountered bugs in the tool that prevented them from running and compiling their code; another had not exposed any program results (e.g., through a `println`) (P9).

Scoops were shorter than slices for task 1 (median = 21 vs. 37 lines) and task 3 (36 vs. 101 lines), and about the same length for task 2 (22.5 vs. 22 lines). Scoops varied from slices in several ways. For task 1, almost all participants removed loop code in order to fetch only one, not many, rows from the database; most participants saved the queried row in a `Book` object, and replaced variable names for column indexes with literals. For task 2, most participants saved the scraped data to a list. For task 3, all but one participant eliminated dozens of lines initializing an email's text by using literal substitutions, or by simply leaving out code that built the message.

Suggestions for improving usability

Participants wanted to format finished code by adding white space to group lines of code by their functionality (P1, P2, P11, P13, P14, P16), and write comments to make code more clear and easier to adapt (P1, P2, P4, P11, P13). Of course, not all participants wanted to comment the code (P6).

Formatting the code as an executable `main` function of a new class was not always seen as necessary. Some participants wanted to create the code as a function that explicitly listed any data dependencies as inputs to the function (P9). One par-

ticipant questioned why they would need to write compilable code, telling us "they're not going to be copied and pasted" (P8), and another suggested they would remove import statements and the class declaration when posting the example (P12), which would cause the example to no longer compile.

Answers to Study Questions

Can programmers extract examples with the intended behavior using CodeScoop? Yes. 16 of 19 programmers successfully extracted example code from existing code in under ten minutes. In each of these cases, the code compiled, ran, and had the behavior authors intended.

How does CodeScoop compare to a standard text editor for extracting example code? Compared to a text editor baseline, CodeScoop's main advantage was its ease of use, providing fixes and suggestions from the original code that participants otherwise had to fix manually. The major feature CodeScoop lacked was direct additions and edits to code.

What code-fixing decisions could CodeScoop make automatically? When extracting code, programmers often responded to CodeScoop's suggestions in different ways. This variation suggests that different contexts and authors prescribe different solutions. Further work is needed to know which solutions are best (if any) for readers of examples.

Do "scoops" offer value over "program slices"? Yes, though with some caveats. Scoops could be more concise than slices. However, when programmers saw alternative suggestions like slices, this caused them to notice other code they wanted to include in the example.

DISCUSSION AND FUTURE WORK

From designing CodeScoop, we gained a deeper understanding of what an example extraction tool can and should support. This could provide a path to future work.

Supporting more example extraction choices

Besides throwing exceptions for error-prone function calls, there were few choices about code extraction that participants made the same way. Decisions about code simplification involved trade-offs that balanced comprehensibility, coding best practices, real-world use cases, and conciseness. At the same time, there was more than one way to achieve such goals as an author: for example, authors could add semantic meaning with a thoughtful variable name or a descriptive comment. Our study shows that these trade-offs play out in different ways for different programmers. While CodeScoop can satisfy some distinct ways of resolving code, what other decision points does it not yet support? We expect one such class of decision points is structural: participants told us they wanted to pull code into parameterized methods, or insert literals as new variables defined at the top of the scoop.

Revealing potentially relevant code from the source program

One participant described going on "auto-pilot" when they interacted with CodeScoop. After comparing their scoop to a program slice, several participants decided there was other code they wanted to include in their scoop. Are revelations about missing code inherent to code extraction? Or can these revelations be avoided with interaction techniques that help programmers discover code they might be missing?

Enabling direct edits while guaranteeing correctness

Almost every participant in the study wrote that one advantage of extracting examples with the text editor is directly editing the example code. This is no surprise—textual edits is a key affordance of all code editors. However, the critique raises an important question: What direct additions and edits should example extraction tools support?

From the study, we believe programmers should at least be able to delete arbitrary lines, add comments, and format whitespace. For some of these edits, it's not clear what the right interaction technique is. Does deleting a line delete the line's dependents? The technology required to support such interactions quickly becomes complex, and we believe there is a rich space of design and engineering challenges waiting to be explored to enable such mixed-initiative example extraction techniques.

Limitations

In the study, participants had only cursory familiarity of the source program. In the intended use case, users will extract examples from their own code, and not code that has been given to them. It could be that participants find CodeScoop easier to use when extracting from their own code. Still, for large code bases, programmers may be just as unfamiliar with the code they work with from day to day, as it could have been written by other programmers or long ago.

The source programs in the study were short: 95 and 135 lines, with all code in one method. We chose these programs because they worked well with the CodeScoop prototype, did not require long comprehension time, and were based on real programs from our own projects. While these programs allowed us to gain insight on how CodeScoop supported example extraction, it's not clear if larger or more complex code will require additional interaction design. There are two hurdles for scooping from larger programs: extending def-use analysis to cover multiple scopes, and coming up with appropriate interaction techniques that can span multiple files while allowing an author to maintain context. Possible solutions may include def-use analysis with interprocedural dataflow (e.g., WALA [2]); and "bubbles"-based code navigation paradigms [5].

CONCLUSION

We developed CodeScoop, a mixed-initiative interaction technique to enable programmers to extract executable code examples from existing code. Our study shows that programmers can use such tools to successfully extract examples. Furthermore, the resulting "scoops" provide value over automatically extracted slices. We believe tools like CodeScoop will ultimately enable programmers to more quickly and effectively share their knowledge through examples.

ACKNOWLEDGMENTS

We thank Jeremy Warner for creating the video figure. This research was supported by the NSF CAREER award IIS 1149799, NSF Expeditions in Computing award CCF 1138996, and an NDSEG fellowship.

REFERENCES

1. Java Debug Interface. https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/
2. WALA: T. J. Watson Libraries for Analysis. http://wala.sourceforge.net
3. Paul Anderson and Tim Teitelbaum. Software inspection using CodeSurfer. In *Workshop on Inspection in Software Engineering (WISE) '01*. 4–11.
4. Titus Barik, Yoonki Song, Brittany Johnson, and Emerson Murphy-Hill. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *ICSME '16*. 211–221.
5. Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola Jr. Code bubbles: A working set-based interface for code understanding and maintenance. In *CHI '10*. 2503–2512.
6. Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: Integrating web search into the development environment. In *CHI '10*. 513–522.
7. Raymond P. L. Buse and Westley Weimer. Synthesizing API usage examples. In *ICSE '12*. 782–792.
8. Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William C. Chu, and Baowen Xu. Dynamic slicing of Python programs. In *COMPSAC '14*. 219–228.
9. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. In *SEFM '12*. 233–247.
10. Shiry Ginosar, Luis Fernando De Pombo, Maneesh Agrawala, and Björn Hartmann. Authoring multi-stage code examples with editable code histories. In *UIST '13*. 485–494.
11. Elena L. Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. Visualizing API Usage Examples at Scale. In *CHI '18*. To appear.
12. Raphael Hoffmann, James Fogarty, and Daniel S. Weld. Assieme: Finding and leveraging implicit references in a web search interface for programmers. In *UIST '07*. 13–22.
13. Ganeshan Jayaraman, Venkatesh Prasad Ranganath, and John Hatcliff. Kaveri: Delivering the Indus Java program slicer to Eclipse. In *FASE '05*. 269–272.
14. Chris Kojouharov, Aleksey Solodovnik, and Gleb Naumovich. JTutor: An Eclipse plug-in suite for creation and replay of code-based tutorials. In *OOPSLA '04*. 27–31.
15. Jens Krinke. Visualization of program dependence and slices. In *ICSM '04*. 168–177.
16. João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *WCRE '13*. 401–408.
17. Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How Can I Use This Method?. In *ICSE '15*. 880–890.
18. Seyed Mehdi Nasehi and Frank Maurer. Unit tests as API usage examples. In *ICSM '10*. 1–10.
19. Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example? A study of programming Q&A in StackOverflow. In *ICSM '12*. 25–34.
20. Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. 2002. What programmers really want: Results of a needs assessment for SDK documentation. In *SIGDOC '02*. 133–141.
21. Christopher Oezbek and Lutz Prechelt. JTourBus: Simplifying program understanding by documentation that provides tours through the source code. In *ICSM '07*. 64–73.
22. Stephen Oney and Joel Brandt. Codelets: Linking interactive documentation and example code in the editor. In *CHI '12*. 2697–2706.
23. Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. 2012. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. Technical Report.
24. Chris Parnin, Christoph Treude, and Margaret-Anne Storey. Blogging developer knowledge: Motivations, challenges, and future directions. In *ICPC '13*. 211–214.
25. T. J. Parr and R. W. Quong. 1995. ANTLR: A Predicated-*LL(k)* Parser Generator. *Software - Practice and Experience* 25, 7 (1995), 789–810.
26. Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *ICSE '12*. 925–935.
27. Martin P. Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE Software* 26, 6 (Nov. 2009), 27–34.
28. Martin P. Robillard and Robert Deline. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (Dec. 2011), 703–732.
29. Marc Sacks. 1994. *On-the-Job Learning in the Software Industry. Corporate Culture and the Acquisition of Knowledge.* Greenwood Publishing Group, Inc.
30. S M Sohan, Craig Anslow, and Frank Maurer. SpyREST: Automated RESTful API documentation using an HTTP proxy server. In *ASE '15*. 271–276.
31. Jeffrey Stylos and Brad A. Myers. Mica: A web-search tool for finding API components and examples. In *VL/HCC '06*. 195–202.

32. Frank Tip. 1995. A survey of program slicing techniques. *Journal of Programming Languages* 3, 3 (1995), 121–189.
33. Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. 2011. How do programmers ask and answer questions on the web?. In *ICSE '11 NIER track*. 804–807.
34. Christoph Treude and Martin P. Robillard. Understanding Stack Overflow code fragments. In *ICSME '17*. 509–513.
35. Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *ICSE '12*. 233–243.
36. Mohsen Vakilian, Stas Negara, Samira Tasharofi, and Ralph E Johnson. Keshmesh: A tool for detecting and fixing Java concurrency bug patterns. In *SPLASH '11*. 39–40.
37. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON '99*.
38. Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. SnipMatch: Using source code context to enhance snippet retrieval and parameterization. In *UIST '12*. 219–228.
39. Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. Bing Developer Assistant: Improving developer productivity by recommending sample code. In *FSE '16*. 956–961.
40. Zixiao Zhu, Yanzhen Zou, Bing Xie, Yong Jin, Zeqi Lin, and Lu Zhang. Mining API usage examples from test code. In *ICSME '14*. 301–310.