

Interactive Program Distillation

Andrew Head



Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-48

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-48.html>

May 15, 2020

Copyright © 2020, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Interactive Program Distillation

by

Andrew Head

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Associate Professor Björn Hartmann, Co-chair

Professor Marti A. Hearst, Co-chair

Professor Koushik Sen

Assistant Professor Joshua Blumenstock

Spring 2020

Interactive Program Distillation

Copyright 2020

by

Andrew Head

Abstract

Interactive Program Distillation

by

Andrew Head

Doctor of Philosophy in Computer Science

University of California, Berkeley

Associate Professor Björn Hartmann, Co-chair

Professor Marti A. Hearst, Co-chair

From snippets to tutorials, programmers rely on sample programs to learn and get work done. The process of creating sample programs, however, can be demanding, limiting the dissemination of programming knowledge. To enhance this process, we introduce the concept of program distillation, methods for its implementation, and usability studies verifying its power. Program distillation is the tool-assisted transformation of existing programs into simpler ones, where key ideas are emphasized, and cruft has been removed.

Three interactive tools are introduced for distilling code snippets, notebooks, and tutorials. Each tool contributes novel interactions grounded in proven program analysis techniques. CodeScoop helps programmers extract snippets from existing code through interactive program slicing and simplification. Code gathering tools let a programmer extract subsets of cells from a computational notebook that reproduce key results. And Torii provides a live programming experience for creating output-rich multi-step tutorials. Studies with users reveal that these tools satisfy important needs, support efficient sample program creation, and provide a level of expressiveness not yet available in today's standard tools.

*To Anna, whose distilled knowledge would fill many
dissertations, each of them worth reading.*

Table of Contents

Table of Contents	ii
List of Figures	vi
List of Tables	viii
Preface	ix
Acknowledgments	xi
1 Introduction	1
Purpose and thesis statement	2
An overview of this dissertation	2
Summary of contributions	3
Research methodology	4
Statement of prior publication	5
2 Background: The design of sample programs	6
Terms	6
How do programmers read programs?	7
Reading order	8
Building mental models of programs	9
Program design choices and their impact on readability	9
How are sample programs used?	11
Why programmers use sample programs	11
The process of finding and using samples	12
What makes a sample program effective?	14
Code snippet design	14

Tutorial design	16
How do authors distill sample programs?	17
The quality of sample programs today	18
Summary	19
3 Related work	21
Tools for authoring sample programs	21
Automated generation of sample programs	21
Literate programming	31
Multi-stage sample authoring	40
Other tools that could support program distillation	43
Efficient code selection	44
Cleaning programs	46
Linked edits to programs, documentation, and outputs	47
Automated program explanation	50
A design space for program distillation tools	51
This dissertation in the design space	54
4 Snippet distillation: Mixed-initiative code selection and simplification	57
Motivation	58
Formative study	60
Method	60
Results	60
Design motivations	62
A demo of CodeScoop	63
Prologue: An unexpectedly useful programming pattern	64
First steps: Initial text selections	64
Mixed-initiative dialogue: Completing the example	65
Implementation	69
Code extraction with the “Flag-Suggest-Resolve” workflow	69
Detecting errors and relevant code	69
Suggesting fixes and code additions	71
Applying fixes to the scoop	72

	Generating an example program from the “scoop” data structure . . .	72
	Implementation specifics and limitations	73
	In-lab usability study	73
	Method	73
	Results	75
	Conclusions	81
	Limitations and extensions	81
5	Notebook distillation: Cleaning messy computational notebooks	84
	Motivation	85
	Design motivations	87
	A demo of code gathering tools	88
	Prologue: A proliferation of cells	88
	Finding the code that produces a result	88
	Removing old and distracting analysis code	89
	Reviewing versions of a result and the code that produced them . . .	90
	Cleaning finished analysis code	91
	Exporting analysis code to a standalone script	91
	Implementation	92
	Collecting and slicing an execution log	93
	In-lab usability study	94
	Method	94
	Results	95
	Conclusions	99
	Limitations and extensions	100
6	Tutorial distillation: Flexible sequencing of snippets	102
	Motivation	103
	Formative study I: Interviews with tutorial authors	105
	Method	105
	Results	105
	Formative study II: Content analysis of two-hundred tutorials	109
	Method	109

Results	109
A demo of Torii	112
Propagating edits from snippets to source programs	113
Propagating edits from code to outputs	113
Splitting, reordering, and copying code	114
Reviewing a simulated reader's code	115
Making localized changes to the code	116
Distributing augmented tutorials	117
In-lab usability study	117
Method	118
Results	120
Conclusions	123
Limitations and extensions	124
7 Conclusions	126
Summary of findings	126
Claim I. Four interactive functions	127
Claim II. Implementation with proven program analysis techniques	128
Claim III. Effective and flexible user experience	129
Remaining challenges and future directions	130
Mixed-initiative program synthesis	130
Authoring tools for explorable tutorials	131
Natural language generation	132
The distillation of scientific discourse and beyond	132
Closing remarks: Humans, compilers, and creativity	134
Bibliography	135

List of Figures

0.1	A snippet from the \TeX program	ix
1.1	An intricate, hand-crafted programming tutorial	1
1.2	Interactive program distillation tools	3
2.1	Four stages of program reading	7
3.1	Classic techniques for presenting programs	22
3.2	A workflow for extracting sample programs from existing programs	24
3.3	An automatically-generated sample program	25
3.4	A flow diagram of a sample usage of a mobile app	29
3.5	A section of a \WEB program and the document generated from it	33
3.6	A schematic of a computational notebook	35
3.7	Types of messes in computational notebooks	36
3.8	A guided tour of a program	39
3.9	Interactive assistance for repairing sample programs	47
3.10	Linked edits of source code clones	48
3.11	A design space of distillation tools, explored	55
4.1	Extracting example code from existing code with CodeScoop	57
4.2	Tool recommendations for improving example extraction	62
4.3	A workflow for iterative correction of incorrect example code	70
4.4	Suggesting fixes and code that complete a “scoop”	71
4.5	Difficulty of making authoring choices with CodeScoop.	76
4.6	A comparison of two participants’ authoring choices	78
4.7	Choices about resolving undefined variables	80
5.1	How code gathering tools extract slices from notebooks	84
5.2	Finding relevant code with code gathering tools	89
5.3	Cleaning a notebook with code gathering tools	90
5.4	Comparing versions of a result with code gathering tools	91
5.5	Implementation of code gathering	92
5.6	Usefulness of features of code gathering tools	97
6.1	An overview of Torii’s approach to snippet execution	102

6.2	Dependencies between artifacts in a tutorial authoring environment . .	107
6.3	Counts of snippets in tutorials, grouped by the tutorial's purpose . . .	110
6.4	Writing tutorials with Torii	112
6.5	Tutorials participants created with Torii	122
6.6	Usefulness of features of Torii.	123
7.1	How program dependencies and linked edits support distillation	129
7.2	Generation of context-relevant explanations of sample programs	133

List of Tables

3.1	A comparison of tools for automated sample program generation	23
3.2	A comparison of tools for literate programming	32
3.3	A comparison of tools for authoring multi-stage samples	40
6.1	How often tutorials contained fragments, duplicated code, and outputs	111

Preface

A carefully presented program

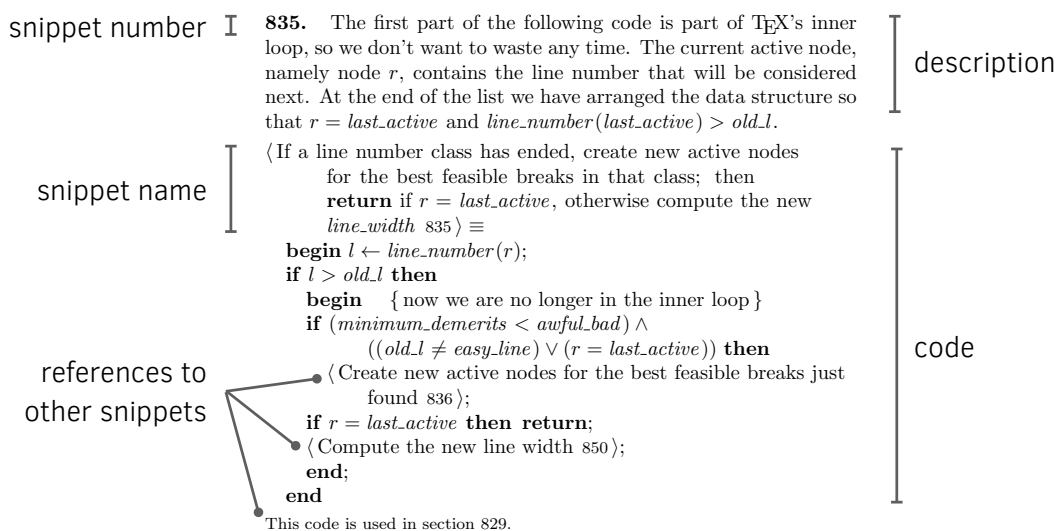


Figure 0.1: A snippet of the TeX program (Knuth 1986)

One of the most beautifully-presented programs is, incidentally, the very same program that was used to typeset this dissertation.

This program is Donald E. Knuth's typesetting system, TeX. The entire source code for TeX can be read in the book, *TeX: The Program* (Knuth 1986). In this book, the program is introduced as a sequence of short snippets of code. Each snippet is accompanied by prose explanations of what the code does. Beneath each snippet are cross-references to related snippets. An index at the end of the book lists each snippet by a descriptive name (Figure 0.1).

To present TeX in this way, Knuth needed new tools. While creating TeX, Knuth (1984) invented the WEB meta-language for authoring programs that could be typeset into books. An author would write a program as a sequence of snippets with explanations. WEB offered authors flexibility in how they split code into snippets.

For instance, any snippet could run code defined in another snippet. And snippets could be listed in any order the author believed was best for helping the reader understand the program. In a program so ordered, Knuth believed:

The complex whole can be understood by understanding each simple part and by understanding the simple relationships between neighboring parts. Large software programs are inherently complex, and there is no “royal road” to instant comprehension of their subtle features. But if you read a well-written WEB program one section at a time, starting with [the first snippet], you will find that its ideas are not difficult to assimilate.

Knuth’s tools, then, provided authors with flexibility to express programs as simple parts and relationships between those parts, and helped authors turn these programs into both machine-executable code and beautiful documents.

At the heart of Knuth’s vision is the belief that tools can help authors present programs in ways that fundamentally change the reading experience. This belief is not unique to Knuth. Rather, it is shared among innumerable researchers, tool builders, and programmers, who have redesigned the medium of the computer program, with inventions from comments to computational notebooks.

This dissertation is founded on that same belief. It begins with the question, *how could a tool help an author present a program that already exists?*

Acknowledgments

To Björn Hartmann and Marti Hearst, I offer my heartfelt gratitude. I cannot fathom how many hours of your time you both have invested in this research and my own development as a researcher. It is from you two that I have learned to pick problems, write carefully, create figures, design slides, draft grants, give feedback, and talk to others about research. I am thankful to have you as my advisors.

To my committee members, I am grateful you took the time to help me learn to be a better researcher. Joshua Blumenstock, thank you for teaching me how to conduct an analytic project, take on problems of societal importance, and craft thoughtful arguments about quantitative research. Koushik Sen, thank you for exposing me to the exciting work of data-driven program analysis for synthesizing programs and texts about them.

To my mentors at Microsoft Research and Google, you have taught me so much about what it means to be an effective researcher and a considerate mentor. If a researcher could have an ounce of Rob DeLine's care in articulating fun conceptual solutions to interesting problems, Steven Drucker's infectious vision for interactive systems, Emerson Murphy-Hill's dedication to inventing fascinating methods, and Caitlin Sadowski's ability to select problems of simultaneous practical and theoretical value, they would be a superhuman researcher with talents unrivaled. I am thankful you took a chance on me.

Thank you to my academic siblings at Berkeley and beyond: Valkyrie Savage, Peggy Chi, Amy Pavel, Philippe Laban, Jeremy Warner, Eldon Schoop, James Smith, David Mellis, Richard Lin, Bala Kumaravel, Ilya Rostovtsev, Michelle Nguyen, Jingyi Li, Forrest Huang, Mitchell Karchemsky, Xiangmin Fan, Wen-can Luo, Xiang Xiao, Teng Han, Phuong Pham, Lanfei Shi, Fred Hohman, and Victoria Hollis: I am grateful to have had you as peers in research and friends. I try and hopefully sometimes succeed in channeling your grace, thoughtfulness, tenacity, and verve into my research and life.

Thank you to those who believed in this work by contributing to it. Matthew Waliman, Nathan Khuu, Jason Jiang, Jocelyn Sun, R. J. Pimentel, Nidhi Kaku-lawaram, Luming Chen, and Kunal Chaudhary: I am blessed that you chose to contribute to this work, impressed that you chose to put up with me as I was learn-

ing how to be a decent mentor, and happy we got to explore new ideas together.

Thank you to my academic micro-mentors at other institution: Dominik Moritz, Sarah Chasins, Philip Guo, Titus Barik, and Cesar Torres. You may not know it, but the advice you gave me and ideas you have planted in my mind continue to resonate with me and weave their way in no small way into the research I choose to do, and how I go about it.

Thank you also to my academic family from the Allen Institute for Artificial Intelligence, Kyle Lo, Sam Skjonsberg, and Dan Weld. You have been tolerant of the time I have spent devoted on this dissertation when we have had new and exciting ideas to explore. You have also taught me how to craft a research project that is simultaneously interesting, stable, and captivating. I am so excited to keep working with you in the next years on ideas that had their humble beginnings in this dissertation but which have now outgrown their domain of programming and now extend into the realms of scientific discourse.

Thank you to my friends who have supported me in writing this dissertation. To my good college friend Cyrus Ramavarapu, thank you for reading the earliest drafts of this thesis before anyone else, and helping me rethink the framing of the dissertation in so many places of consequence. Katie Stasaski and Nate Weinman, this dissertation would have been much more difficult to write had it not been for your regular constant words of encouragement, humorous chat messages, probing questions, and revitalizing weekend tea parties.

Elena Glassman and Gustavo Soares, thank you for being here to breathe life into my research with encouragement and excitement during a formative time in my Ph.D. I didn't know it at the time, but I needed you both. Elena, I aspire to think as big as you, as ceaselessly. Gustavo, I didn't know mortals could create such beautiful algorithms until I saw you create one in the spring of 2017.

Thank you to Jingtao Wang, my first mentor in human-computer interaction at the University of Pittsburgh. Jingtao, there is no question I wouldn't be here today if you had not replied to my email asking if you had room for an undergraduate researcher in your group to study serious video games. If you had not had faith in my potential as a researcher, I would not be at UC Berkeley, and I would not be writing this dissertation. You told me many years ago that I should do with my life what I would do even if I had all the time and money in the world. I think you'd be happy to know that I am. I try every day to make you proud.

Finally, thank you to my wife, Anna. You celebrated with me when good news came. You helped me figure out how to take care of myself during the busiest times of this dissertation. You laughed at my jokes, most of which really did not deserve anyone's attention. You offered feedback on ideas, talks, and paper drafts in the

late hours of the night. And you joined me in discussions of human-computer interaction and programming languages over breakfasts and dinners. Thank you for enjoying the cozy little niche of research in this dissertation with me, and for building an exciting life with me outside of it.

Chapter 1. Introduction

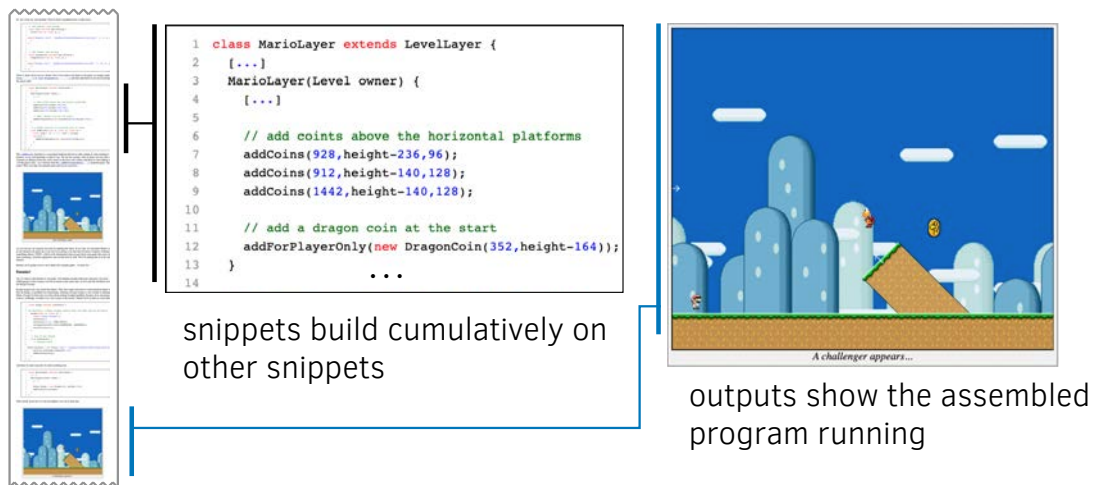


Figure 1.1: An intricate, hand-crafted programming tutorial. To create this tutorial, an author must have created many versions of the same source program, created outputs for each version, and carefully styled the snippets. The pictured tutorial is “Let’s make a Mario game” by Kamermans.

HUMANS are capable of producing marvelous instructions for one another. For example, take the programming tutorial shown in Figure 1.1. The tutorial teaches a reader how to create the popular Super Mario side-scrolling game by writing about 500 lines of code using the Processing language. The code is split into bite-sized snippets each about 20 lines long. Each snippet is carefully styled, including just enough context to tell the programmer where to place the new code, and cutaways (i.e., “[...]”) to hide code previously added. After every few snippets, the reader can test out the program in each of 22 different interactive versions of the Mario game at various stages of construction.

While this tutorial is impressive, what’s more impressive is the effort that must have gone into preparing its code. Tutorials like this are created by authors transforming existing programs into snippets and accompanying outputs. Producing these snippets and outputs involves *selecting* code from large and tangled programs, *simplifying* the code to make it more readable, *supplementing* it with text

and outputs, and *sequencing* it into a series of snippets. The above tutorial contains 41 snippets and 22 outputs. One can imagine it took quite some time to extract these snippets, clean their code, and produce outputs using them.

Program distillation is the transformation of existing programs into sample programs like tutorials and code snippets. It is the process by which tutorials like the one above are brought into existence. The purpose of distillation is to bring out the key ideas in a program, while eliminating cruft. All around the world, hobbyists, technical writers, professional developers, and teachers invest considerable effort in distilling programs for others to read and reuse.

Purpose and thesis statement

While programmers benefit from mature tools for many activities like debugging and reviewing code, authors distill programs with a mish-mash of general-purpose tools. Like debugging, distillation entails a common set of subtasks and pain points that authors encounter when performing those tasks. Also like debugging, a well-designed tool can alleviate these pain points, permitting authors to offload cognitive load to their tools and instead focus on higher-level design goals.

The purpose of this dissertation is to answer the question, *How can tools help authors distill existing programs into sample programs?* My thesis is that

Authors can transform existing programs into sample programs more efficiently and flexibly when aided by interactive tools for selecting, simplifying, supplementing, and sequencing code.

An overview of this dissertation

This dissertation introduces distillation tools, a class of programming tools designed to help authors transform existing programs into sample programs (Figure 1.2). It comprises the following chapters:

Chapter 2 begins with a review of background knowledge on the topic of program distillation. The chapter offers guidelines for how to present sample programs effectively, grounded in empirical studies of programmers. It also introduces the key tasks of program distillation, consisting of selecting code, simplifying it, supplementing it with other code and assets, and sequencing it.

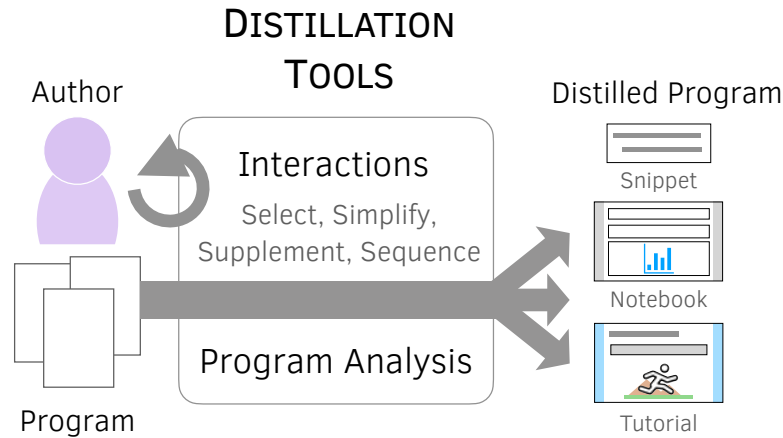


Figure 1.2: Interactive program distillation tools. With distillation tools, authors have interactive affordances for efficiently *selecting* code, *simplifying* code, *supplementing* the code with outputs and explanations, and *sequencing* it into a series of snippets.

Chapter 3 reviews the related work. It first surveys how prior tools have helped authors create and present sample programs. It then considers other tools and techniques from the fields of human-computer interaction, software engineering, and program analysis, that may serve as useful components of future distillation tools. A morphological design space of distillation tools is introduced.

Chapters 4–6 constitute the novel research contributions of this dissertation. Together, they represent an initial exploration of a larger design space of distillation tools. In isolation, each chapter introduces a new tool for distilling a specific class of programming artifact—a snippet, a notebook, or a tutorial. The contributions of each chapter are summarized in the next subsection.

Chapter 7 concludes this dissertation, summarizes findings, and sets an agenda for future work in program distillation.

Summary of contributions

The primary contributions of this dissertation are the design, engineering, and evaluation of three novel interactive systems. These systems support:

Mixed-initiative extraction of concise, executable snippets from existing code via incremental program slicing and simplification of source programs. A controlled lab study showed that programmers could extract snippets more efficiently with this system than with a comparison tool (Chapter 4).

Rapid cleaning of messy computational notebooks via the selection of variables or outputs and program slicing of an execution history. A qualitative usability study showed that analysts found the system useful for cleaning notebooks and writing analysis code, and discovered new ways to use them, like generating personal documentation and lightweight versioning (Chapter 5).

Construction of flexibly-organized programming tutorials via live, linked editing of source programs, snippets, and outputs. The system provides authors flexibility in how they split code into snippets and how they order those snippets. A usability study showed that authors could use the system to write simple tutorials with flexibility not present in standard code editing tools (Chapter 6).

Additionally, this dissertation contributes two formative studies of tutorial authoring in Chapter 6. These studies reveal that authors face a unique challenge of keeping collections of related programming artifacts consistent with each other as they write and revise tutorials.

Research methodology

The primary contributions of this dissertation are *artifacts*. Artifacts are one of the more common types of contributions within the discipline of human-computer interaction. The purpose of research involving artifacts is to reveal design possibilities, yield generalizable insights, and capture an understanding of the design problem in the artifacts (Wobbrock and Kientz 2016).

Interactive systems like those in this dissertation are often evaluated holistically based on a variety of factors such as importance, generality, and impact on solution viscosity (i.e., users' difficulty accomplishing tasks) (Olsen Jr. 2007). Each of these three criteria are discussed throughout this dissertation. The importance of the systems comes from their reach: the systems could one day support millions of authors in producing sample programs, and millions more readers who would read those samples. The generality of the tools is demonstrated through their implementation in existing code editors.

The ability of the tools to reduce solution viscosity is evaluated through in-lab usability studies with users. Qualitative usability studies were preferred to controlled studies, given their effectiveness in yielding insight about user expectations, strategies, and preferences. That said, small-scale controlled studies were employed to evaluate two of the tools (Chapters 4 and 6).

The systems were designed iteratively with input from users. Initial ideas for the systems came from formative studies of authors and programs they created. For each system, there are dozens of sketches, paper prototypes, and interactive

prototypes that were created and discarded as a more nuanced understanding of user needs, capabilities, and technological limitations was attained.

Statement of prior publication

Chapters 4, 5, and 6 have previously appeared as conference papers in the proceedings of the SIGCHI Conference on Human Factors in Computing Systems (Head et al. 2018; Head et al. 2019; Head et al. 2020).

All of my co-authors on these articles have provided their consent for the articles to be reproduced in this dissertation. Much of this dissertation, therefore, represents the gracious contributions of my co-authors. Elena L. Glassman drafted the introduction and figures of what is now the chapter on snippet extraction (Chapter 4). She worked with me tirelessly to scope the project and articulate a vision of how mixed initiative tools could help with snippet creation.

The idea for notebook distillation (Chapter 5) arose in my very first conversations with Rob DeLine and Steven M. Drucker. As I now know, the idea germinated in Rob and Steven’s minds years before we met. After our formative research, it was clear this idea would be impactful, and a perfect union of our interests. Rob wrote the initial version of the program dependence analysis code that evolved into the code gathering tools’ backend. Rob, Steven, and Titus Barik piloted and ran the usability study for notebook distillation at Microsoft Research.

When designing Torii (Chapter 6), Marti A. Hearst helped design the protocol for the content analysis of tutorials. Nidhi Kakulawaram and Luming Chen piloted this protocol, and contributed to its refinement. Jason Jiang undertook the painstaking work of serving as the second analyst for all 200 tutorials, in just three weeks. Then, James Smith conducted nearly all of the usability study sessions, surfacing key stories from each one.

The guiding advice of my mentors Björn Hartmann and Marti A. Hearst can be found throughout each chapter within this dissertation, for aspects big and small, from project motivations to system names, figure design, and word choice.

Chapter 2. Background

The design of sample programs

Designing a sample program is an activity involving hundreds of small decisions. Among other things, an author must decide what code to share, and how to rewrite it so that it can be easily read, understood, and reused.

The purpose of this chapter is to provide an empirically-grounded understanding of what we know about how sample programs *should be presented*, and how they *are authored* today. This chapter briefly surveys what is known about how programmers read programs, how they make use of sample programs specifically, and how authors write sample programs. The chapter culminates in a summary section, consisting of guidelines for effective sample program design, and a set of author needs for future tools to support.

To begin, this chapter defines the terms used in the rest of this dissertation, such as *author*, *reader*, and *sample program*. These terms have a nuanced meaning in this dissertation, and are therefore defined precisely below.

Terms

The purpose of this dissertation is to help programmers create sample programs. **Sample programs** (abbreviated “samples”) are programs that are written to be reference material for other programmers.

Because they are designed to serve as reference material for other programmers, samples are distinct from other types of programs a programmer might write. Unlike prototype programs, which are often messy and thrown away, samples are meant to be read by others and may have a long lifetime. Unlike production programs, which are built to be maintainable and robust, samples are often written to be simple, short, and readable at the expense of robustness.

Those who produce samples are **authors**, and those who read them are **readers**. Readers have three reasons to read samples: to learn, to gain design inspiration, and to reuse code.

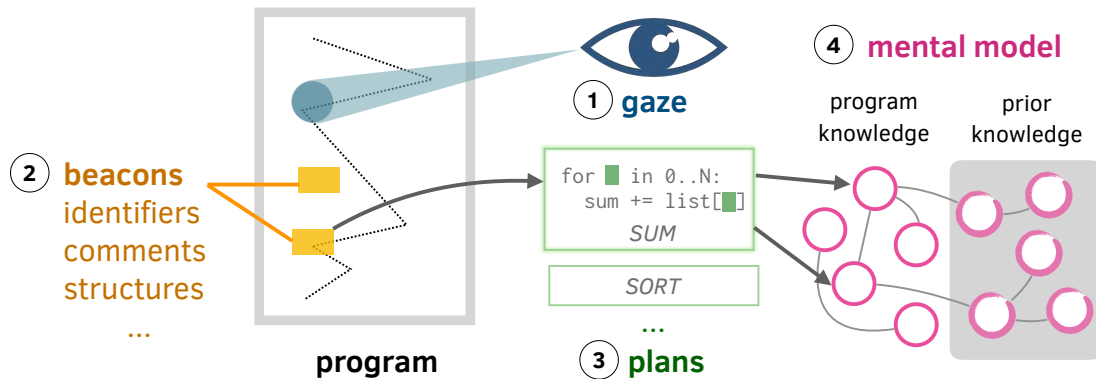


Figure 2.1: Four stages of program reading. As programmers read programs (1), they find beacons (2). These beacons help them recall plans, or programming patterns they already know (3). A mental model of the program is built by integrating knowledge about the program with prior knowledge (4).

Two types of samples are considered in this dissertation. **Code snippets** (abbreviated “snippets”) are short code listings about a dozen lines long, and rarely longer than one hundred lines long. **Tutorials** are documents consisting of a sequence of snippets, interspersed with prose explanations of the snippets. The tools described in this dissertation help authors create both snippets and tutorials.

Authors use **code editors** to create samples. Chapter 5 focuses on the distillation of samples within a specific type of code editor called a computational notebook. A **computational notebook** (abbreviated “notebook”) is a code editor consisting of a sequence of code cells. Authors write code in these cells and execute each of them one at a time. The outputs from running the code in a cell get embedded in the notebook next to the code that produced it.

How do programmers read programs?

Any one program can be written in innumerable ways. How can an author write a sample program so that it can be easily read? How should the code be organized? What names should the variables be given? Should it include comments? This section reviews research about how programmers read programs and discusses implications for the design of readable programs. A visual summary of the process of reading programs is shown in Figure 2.1.

Reading order

Programmers do not read programs linearly. Busjahn et al. (2015) conducted an eye-tracking study where they asked programmers to read programs and answer comprehension questions about them. The programmers' gaze patterns were compared to two idealized orderings: one where statements were ordered by line number, and another where they were ordered by control flow, i.e., the order in which the program would execute them. It was observed that reading order was better predicted by control flow than by linear order.

Reading order varies from one programmer to another. Busjahn et al. (2015) found that student programmers read code in a more linear order than professional programmers. In another eye-tracking study, Uwano et al. (2006) observed student programmers as they inspected programs for bugs. While many of the programmers began inspecting the code by scanning it in its entirety, scanning patterns varied. Some participants scanned the program more quickly than others. At least one participant scanned the entire program twice.

Two general approaches for reading programs have been identified in the program comprehension literature: *systematic* and *as-needed*. Littman et al. (1987) identified these two approaches when observing programmers as they modified a program. When reading *systematically*, a programmer would execute the program in their mind and seek an understanding of causal relationships between disparate parts of the program. In contrast, when reading *as-needed*, the programmer would read as little of the program as possible, locating just the parts they needed to understand in order to modify the program. Those who read the program *as-needed* were less successful at modifying the program. Littman et al. saw no relationship between a programmer's experience and their approach to reading.

As they read a program, it is theorized that programmers rely on the program's surface features to develop and confirm hypotheses about what the program does. Brooks (1983) named the features that programmers use to recognize known structures and operations *beacons*. One example of a beacon is a section of code that swaps two elements of an array within a loop, which could indicate that a sort operation is being performed. In Brooks' theory, comments, identifier names, indentation, and external documentation can all serve as indicators that provide beacons to help programmers recognize known structures and operations.

When modifying a program, a programmer will read some parts of the program to orient themselves, and other parts to modify the program. Koenemann and Robertson (1991) observed programmers as they modified a program. They imposed a unique constraint, telling participants they could only access one piece of information at a time, i.e., one snippet, output, or piece of documentation. Koen-

emann and Robertson found that programmers accessed outputs and high-level program descriptions early in the programming task because of their *strategic relevance* to the task. These resources helped the programmer understand the goals and organization of the program. Later, programmers focused almost exclusively on the code snippets they were modifying and descriptions of those snippets, which were of *direct relevance* to their modification task.

Building mental models of programs

As programmers read programs, they build up mental models of what the program does and how it does it. The design of the program can influence whether those models are correct, and how efficiently they are built. It is believed that programmers look for *plans* in programs, that is, program fragments that represent stereotypic programming action sequences. For instance, one plan might be the computation of a running total in a loop (Soloway and Ehrlich 1984).

Soloway and Ehrlich (1984) found that a programmer's ability to recognize plans in a program depends on how well a program follows *rules of discourse*. One rule of discourse is that a variable should be named by its role within a plan. Another rule is that if code is included in a program, it should be executed. When these rules of discourse are violated, for instance by assigning unexpected names to variables, Soloway and Ehrlich found that programmers had difficulty recognizing plans and performing simple code completion tasks.

The result of a close reading of a program is a nuanced and multi-faceted mental model. Wiedenbeck et al. (1993) found that programmers who carefully studied a program understood its hierarchical structure, developed a mapping between the program's goals and its code, recognized common programming patterns, connected the program with their prior knowledge, and grounded their understanding of the program in textual details. Pennington (1987) found that the strength of a programmer's mental models depended on the time spent studying a program. Programmers developed procedural models of a program's control flow first, before functional models of its goal hierarchy. Functional models of the program were strengthened when the programmers were asked to modify the program.

Program design choices and their impact on readability

Does the structure of a program affect its readability? How about the style of identifier names? In a recent online survey, Tashtoush et al. (2013) found that programmers expect that many features of a program to influence its readability. The literature provides evidence showing that many of these features do indeed

impact programmers' ability to understand, debug, and modify a program. Here, we briefly summarize findings from the literature.

Module structure. Kinicki and Ramsey¹ evaluated the impact of module structure on program comprehension. Code was divided into modules, either according to function, randomly, or not at all. Comprehension of programs was best when code was split into modules according to their function. Boehm-Davis et al. (1992) asked programmers to modify programs organized in one of three ways: in-line code, functional decomposition, or object-oriented. For student programmers in the study, the programs organized by functional decomposition were the fastest to modify, and the easiest to find information in. No effects were observed for professional programmers in the study.

Statement order. Letovsky and Soloway (1986) observed programmers as they modified programs containing *delocalized plans*, or programming plans that were scattered across different parts of the program. Programmers had trouble modifying programs with delocalized plans, particularly if they read the program according to an *as-needed* approach rather than a *systematic* approach. Dunsmore et al. (2000) found that programmers were less likely to be able to find bugs that are caused by interactions across scattered lines of code. This implies that authors should take care to group related statements of code close together.

Whitespace. Norcio (1982) asked student programmers to fill in the blanks in programs with selected lines removed. Programs varied based on whether they were indented and whether comments all appeared at the top or instead interspersed with the text. Both indentation and interspersed documentation had a positive effect on a programmer's ability to fill in the blanks. Miara et al. (1983) asked student programmers to answer comprehension questions about a program formatted with differing numbers of spaces per indent. Students answered questions most accurately when the program was formatted with four spaces per indent, rather than 0, 2, or 6 spaces. Oman and Cook (1990) conducted four studies of how the format of a program affects a programmer's ability to modify and answer questions about the program. Many features of programs were changed at once, including separating control structures by blank lines, chunking related clauses, and highlighting section headings. The design of the studies made it impossible to determine the effect of each type of change. That said, all four studies showed that the proposed changes led to improvements in task performance or efficiency.

Identifier names. Lawrie et al. (2006) conducted an online study wherein programmers were asked to write descriptions of programs that varied based on the brevity

¹ As described by Shneiderman and Mayer (1979).

of identifiers. Programs included identifiers that were either full words, abbreviations, or single letters. Programmers' descriptions were most accurate when the program contained full-word identifiers. Binkley et al. (2013) conducted a series of five experiments on identifier style, concluding that identifiers presented using camel-case (i.e., `getNextPath`) rather than underscores (i.e., `get_next_path`) led to better comprehension, particularly for beginner programmers.

Comments. Many studies have measured the impact of comments on program comprehension. Nielebock et al. (2019, Table 1) provide a recent review. The studies in the review suggest that comments can help beginner programmers answer questions about code, and complete programming tasks. Nielebock et al. conducted an online study of their own where hundreds of programmers, mostly professionals, performed short programming tasks on small programs, with and without comments. The presence of comments did not impact programmers' success or efficiency in completing the tasks. This suggests that while comments may aid comprehension, it depends on the comment, the reader, and the task.

Colors. When important elements of a program are given a font or background color, programmers can better answer questions about the program (Rambally 1986), optimize the program (Tapp and Kazman 1994), and complete specialized tasks like locating code within preprocessor directives (Feigenspan et al. 2013). One common form of coloring in code editors is syntax highlighting, where tokens are colored based on their role in the program grammar. Hannebauer et al. (2018) provide a recent review of research on the impact of syntax highlighting on program comprehension, noting that prior research has yielded contradicting results. Their own study did not show any effect of syntax highlighting on student programmers' completion of programming tasks. Altogether, the above research suggests that coloring helps programmers read and locate code, even if syntax highlighting does not consistently impact the programming experience.

How are sample programs used?

To design an effective sample program, it is useful to understand why a reader uses samples, and how programmers find and use them.

Why programmers use sample programs

Readers use sample programs to learn, to gain design inspiration for their own programs, and to find code to reuse in their programs.

Learning. In a field study of programmers learning on-the-job, Sacks (1994) reported that programmers “learn more and do more work by copying and following examples and by working their way through exercises than by any other single activity.” When learning new programming languages and tools, programmers would follow along with tutorial materials, complete the exercises, and then use the samples to guide their own practice. In more recent studies, when programmers have been tasked with learning about a new technology or tool, they often start by searching for tutorials (Brandt et al. 2009), or finding samples that they can use to scaffold the creation of their own programs (Sillito and Begel 2013). Even in industry, tutorials like “Codelabs” at Google make up an important part of the on-boarding materials (Johnson and Senges 2010).

Design. Samples help programmers refine the design of their programs. Burkhardt and Detienne (1995) conducted an in-lab study where programmers were asked to design a new software system. Throughout the study, programmers looked at existing designs to clarify their goals, revise their solutions, and find constraints for the design problem they had not considered before. Wu et al. (2019) analyzed how programmers reused code from Stack Overflow, a popular programming question-and-answer site. Sometimes, programmers consulted samples as reference implementations, without copying any code from the sample. The act of *creating* samples may help a programmer design programs as well. In their textbook on program design, Felleisen et al. (2018) advocate that programmers create samples as they design a program to develop a specification of its behavior. Beck (2001) claims that writing tests for a program that has not yet been written can help a programmer scope their implementation and simplify their designs.

Reuse. Programmers reuse code from samples as building blocks for their own programs (Sacks 1994; Brandt et al. 2009). They consult samples to validate the correctness of the syntax used in their own programs (Neal 1989; Sacks 1994). Evidence of the reuse of samples is pervasive in public repositories of source code. Baltes and Diehl (2019) conducted several studies of how code is cloned from samples on Stack Overflow into public source code repositories. They found that, depending on the repositories considered, between 3.3% and 11.9% of repositories contained clones of code from Stack Overflow.

The process of finding and using samples

An effective sample program is one that can be found, understood, copied, modified, and debugged. This subsection describes how programmers find and use samples, from search to integration.

Step 1. Search. When a programmer needs a sample, they will often search for one using a general-purpose web search engine (Brandt et al. 2009) or a search engine that indexes their organization’s code (Sadowski et al. 2015). Programmers search for samples for many types of tools and technologies, like third-party libraries and services, build systems, and front end development tools (Xia et al. 2017). For some search engines, searches for samples account for as much as 20% of programming-related queries (Hoffmann et al. 2007).

After submitting their query, the programmer might open many pages from the search results before inspecting any one of them (Brandt et al. 2009). The question then is which sample to use. The programmer assesses whether to use a sample based on the page’s cosmetics, like whether the code has syntax highlighting, or whether the page contains ads (Brandt et al. 2009). They may also consider the author’s credentials, the availability of working code demos, the match between the programming language of the sample and programmer’s own code, and the “digestibility” of the information (Dorn and Guzdial 2010). Surface features of a sample may be deceiving. For instance, when asked to make a head-to-head choice between samples, programmers may choose those that have detailed, well-written explanations, even if the code is fundamentally flawed (i.e., contains critical security vulnerabilities) (Linden et al. 2020).

Step 2. Copy. Once a programmer has found a promising sample, they copy it into their own program. What is most interesting about this step isn’t that code is copied, but what *isn’t* happening as the programmer copies it. Specifically, programmers often won’t read the code carefully, or the surrounding text. Instead, they will instantly start trying to modify it (Brandt et al. 2009). Explanations around the sample program may be ignored (Anderson et al. 1984; Brandt et al. 2009). Flaws in the code get copied along with the code (for one example, see Pirolli and Recker 1994). This tendency to avoid looking carefully at the contents of code to be reused has been called *comprehension avoidance* in the setting of a programmer reusing their own code (Lange and Moher 1989), and the term feels apt for this setting as well. While not every programmer will ignore the code and text for every sample they copy, this behavior has been observed often enough that it should be considered a major usage pattern.

Step 3. Modify as needed. After programmers copy code from a sample, they modify it until it fits into their own program and runs correctly. The process of modifying a program involves repeated editing, execution, and debugging. Rosson and Carroll (1996) have described the process as “debugging into existence”: rather than carefully planning how to integrate the copied code into their program, the programmer allows their tools to guide them to a working program by pointing out one error at a time. By the time code from a sample has been integrated, it will

have been modified in many ways. Perhaps the cosmetics of the code have been changed, or the functionality of the code might have been altered (Wu et al. 2019). Zhang et al. (2019) estimate that 88% of clones from Stack Overflow into GitHub projects required 23 or more modifications, for the purposes of fixing compilation errors, refactoring, and customizing logic.

Implications of empirical research about sample program usage

One implication of the studies above is that to make a sample program effective, an author needs to think about more than readability. First, the appearance of the sample matters, because a reader might not use a sample if it does not appear attractive and well-explained. Second, the sample should be easy to modify, because the reader will often want to integrate it into their own code. Third, the sample should be robust, because the reader may not read the program and the accompanying explanations thoroughly.

These studies describe how programmers reuse samples, and not how they learn from samples. What can an author do to help readers learn from samples? Anderson et al. (1984, pp. 90–93) provide a vivid account of a programmer learning recursion from a textbook. An incident is described where a programmer used a sample as a template for doing an exercise. By modifying the sample, the learner acquired new procedural knowledge that they then were able to apply almost immediately and efficiently to solve a subsequent problem. In other words, the programmer learned from the sample by using it to solve problems. The implication of Anderson et al.’s observations is that authors can help readers learn from samples by providing opportunities to use them to solve problems.

What makes a sample program effective?

Prior research on sample programs has yielded explicit guidelines on how they should be designed. These guidelines are reviewed here.

Code snippet design

To learn what makes snippets effective or not, researchers have asked programmers to describe their experiences using samples in interviews (Nykaza et al. 2002; Robillard and Deline 2011), surveys (Robillard and Deline 2011; Buse and Weimer 2012), and diary studies (Sillito and Begel 2013). They have also analyzed the content of high-quality snippets (Buse and Weimer 2012; Nasehi et al. 2012). These

studies have revealed five traits of effective snippets. Some of the traits conflict with each other, suggesting that readers require different types of samples for different purposes.

Conciseness. Two content analyses of high-quality snippets showed that they are often concise. Buse and Weimer (2012) found that samples in the Java SDK were 11 lines long on average, with a median length of 5 lines. Nasehi et al. (2012) observed that samples in accepted and highly-voted answers on Stack Overflow were often shorter than the samples in other unaccepted or less-highly-voted answers. In both content analyses, unnecessary code was replaced with placeholder methods like “`processVariable(var)`,” comments like “`// compare objects`,” and ellipses like “`int glyphIndex = ...`” When conducting a needs assessment for SDK documentation, Nykaza et al. (2002) learned that programmers often desired short samples containing 10 lines of code or less.

Simplicity. In interviews and surveys with professional programmers, Robillard and Deline (2011) report that programmers sometimes found long samples hard to understand, because the samples became “tangled” with code that accomplished multiple disparate goals. Buse and Weimer (2012) surveyed students in a software engineering course, asking them to describe what factors were important in good samples. Among other factors, respondents reported that samples should show “the most basic version of the problem,” and include simple and understandable variable names. Another way to achieve simplicity is to split up the code into small chunks. In their content analysis of Stack Overflow answers, Nasehi et al. (2012) observed that accepted and highly-voted answers often show the code split into multiple chunks, where each chunk is explained separately.

Non-triviality. A sample should not be so simple as to do nothing useful at all. It must show how to do something useful. In their needs assessment, Nykaza et al. (2002) found that programmers wanted both simple *and* advanced samples. In interviews and diary studies, Sillito and Begel (2013) learned that programmers consult blog posts and Stack Overflow to learn how to use their tools in ways that are not covered in official programming documentation. These sources helped programmers understand “ways to use it you may not have thought of” and led to “a-ha moments.” Robillard and Deline (2011) suggested that the ideal size of a sample is a single programming pattern, rather than a single API call.

Completeness. When learning on their own, programmers write programs by making use of “vanilla examples” that they can be sure will work (Sillito and Begel 2013). In their needs assessment, Nykaza et al. (2002) heard from programmers that they want robust samples that can be used as is, or modified and incorporated into larger applications (Nykaza et al. 2002).

Adherence to best practices. Robillard and Deline (2011) report that their informants perceived samples to be not only demonstrations of how to do something, but as recommendations about how *best* to do something. Some informants believed that samples were the best type of documentation for describing best practices. Informants would choose which samples to use based on whether the samples seemed sufficiently authoritative, credible, and recent to demonstrate best practices. In their content analysis of Stack Overflow answers, Nasehi et al. (2012) found that accepted and highly-voted answers often explicitly described the limitations of the code, such as performance issues and security risks.

Tutorial design

Programming tutorials are, fundamentally, just one specific type of tutorial. Researchers have studied how to design effective tutorials for decades, and that research applies to programming tutorials as well. van der Meij et al. (2009) reviewed decades of research on tutorial design as of 2009. The review surfaced several key findings backed by empirical research. First, instructions can be made more effective if they follow the minimalist instruction paradigm, a paradigm described at length in Carroll's (1990) book *The Nurnberg Funnel*. In this paradigm, instructions help readers accomplish their own tasks, leverage their existing problem-solving strategies, and recover from errors. Second, instructions should provide both procedural information about what actions need to be performed, as well as declarative information which includes supportive information about the general workings of a program and tips and tricks. Third, if an author wants to encourage readers to try out features, this is best accomplished by providing exercises instead of simply asking readers to experiment with the software "on-their-own."

Further research in the programming domain has led to recommendations for pedagogy and teaching problem-solving process in programming tutorials:

Pedagogical best practices. Kim and Ko (2017) introduced 24 dimensions for evaluating the pedagogical effectiveness of a programming tutorial. Dimensions included whether the tutorial helps learners select appropriate learning material, provides targeted feedback, aids transfer learning, and helps learners seek answers to their questions beyond the tutorial. Kim and Ko selected these dimensions based on research in the learning sciences. Perhaps authors should consider these dimensions when brainstorming how to improve the learning experience of their tutorials.

Teaching problem-solving process. Linn and Clancy (1992) described *case studies* as a type of instructional material that helps beginner programmers learn expert-level program-solving process. Case studies comprise a programming problem, a

description of the process used by an expert to solve the problem, an expert's code, study questions, and test questions. One of the key motivations behind case studies is to help learners build multiple representations of solutions like an expert often has. A case study thus includes and links together pseudocode, sample programs, illustrations, verbal descriptions, implementations, testing information, and debugging information. Linn and Clancy found that when learners were provided code with expert commentary like that from the proposed case studies, they learned significantly more about program design.

How do authors distill sample programs?

Recently, research has offered insight into how authors today create sample programs from existing programs. Studies have included interviews with authors (Dagenais and Robillard 2010; Ford et al. 2016; MacLeod et al. 2017; Mysore and Guo 2017), surveys of authors (Parnin et al. 2013), observations of programmers creating sample programs (Ginosar et al. 2013; Ying and Robillard 2014), and content analyses of sample programs (Tiarks and Maalej 2014). These studies indicate that authoring sample programs involves four tasks.

Selecting. Coming up with ideas for samples can be difficult, as authors realize they need to come up with samples that are realistic and not too contrived (Dagenais and Robillard 2010). Parnin et al. (2013) found that programming bloggers often blogged about their own coding experience. Once a blogger had an idea for a sample, they would need to find, extract, and format code. Bloggers did this manually. Bloggers often prepared posts several days after their initial experience, which affected their ability to recall and recreate their experience.

Simplifying. Once an author selects code from an existing program, they *simplify* it to make it readable and reusable. As noted by a blogger in Parnin et al.'s (2013)'s survey, existing programs include unnecessary dependencies and customized business logic that need to be removed. Ford et al. (2016) interviewed programmers about barriers they faced to contributing questions and answers on Stack Overflow. Informants reported that their programs sometimes contained company secrets, and were too specific, long, or detailed to understand. The process of stripping their program of proprietary information and unnecessary code was enough of a burden to be a barrier to contributing to Stack Overflow. Ying and Robillard (2014) found that when programmers were asked to create three-line summaries of programs, some common operations for formatting the code included renaming identifiers, hiding code, and introducing indentation.

Supplementing. Authors *supplement* samples by improving the quality of the code, explaining the code, and producing visual aids. Tiarks and Maalej (2014) analyzed hundreds of tutorials, finding that typically, tutorials contain thousands of words, as well as images. Mysore and Guo (2017) interviewed teaching staff for a programming course, finding that the staff produced rich tutorials that contained PowerPoint slides, video clips, and commands. The PowerPoint slides included screenshots showing expected visual outputs. The staff also produced skeleton starter code, helper scripts, and validation scripts. Creating these supplemental materials could be time-consuming, and it could be difficult to keep all of these materials updated and in sync.

Sequencing. To present a sample effectively, authors might wish to *sequence* it, showing multiple versions of that program as it gets built up from nothing. Common ways to sequence a program are recording a screencast as one writes the program, creating multiple “stages” or versions of the same program, or splitting it into snippets. When an author *sequences* a program, they implicitly create multiple versions of it. MacLeod et al. (2017) interviewed creators of programming screencasts, finding that they often prepared for a screencast by developing an outline, and sometimes debugged their programs live during the screencast. Ginosar et al. (2013) found that when programmers create samples with multiple stages, they often made changes that they wished to propagate to the other stages, like fixing a bug or improving the program’s presentation.

Each of the studies above focused on a specific type of author, producing a specific type of sample. Much remains to be understood about how samples are authored and how tools can support authors. Therefore, Chapter 4 contributes an observation study of how programmers select and simplify their code to make samples, and Chapter 6 contributes an interview study revealing how accomplished tutorial authors supplement and sequence their programs.

The quality of sample programs today

There is much evidence that, on the whole, programmers succeed in using the samples at their disposal. As evidenced by the studies described in this chapter, programmers regularly use samples to learn, to design, and to implement programs of their own. Most questions on Stack Overflow receive answers that are upvoted and accepted by the asker (Mamykina et al. 2011). Some tutorials personalize content, provide targeted and immediate feedback, and help programmers transfer what they learned beyond the current tutorial (Kim and Ko 2017).

That said, studies have also shown that many samples do not meet expectations. Programming tutorials found on the web often do not provide feedback,

support personalization, or support transfer learning in ways that could be helpful to readers (Kim and Ko 2017, Table 1). In a content analysis of programming textbooks, Lin and Wu (2007) found that samples used in textbooks were often dry and too trivial to provide meaningful instruction about how to solve programming problems. Sometimes, snippets found online cannot be compiled or parsed (Yang et al. 2016), contain violations of the libraries used, (Zhang et al. 2018), and are considered to have inadequate organization, unhelpful naming, unexplained rationale, and clutter (Treude and Robillard 2017). Samples are sometimes missing for popular public APIs (Parnin et al. 2012).

The tools in this dissertation are designed to help programmers produce clean, complete, well-formatted sample programs from their existing programs. It is hoped that by automating the clerical work in the tasks of selecting, simplifying, supplementing, and sequencing code, the tools can let authors focus on creating programs that are more readable, usable, and meaningful.

Summary

The above reviews of the literature can be distilled into a set of guidelines for how sample programs might best be presented:

1. ***Reduce program fragmentation.*** A programmer may find it difficult to debug programs when the code relevant to the bug is scattered across different parts of the program. Help readers understand and debug the sample program by keeping related statements close together.
2. ***Use familiar programming idioms.*** A reader’s ability to understand a program may depend on their ability to see programming plans that they already know. Help the reader recall plans by using familiar program structures and variable names, and by providing clues in the comments.
3. ***Support reader needs.*** Readers consult samples to learn, to improve their program’s design, and to reuse code. Readers desire sample programs that show non-trivial programming patterns, “vanilla” use cases, and best practices. Sample programs cannot provide a perfect solution to all of these needs at once. Authors should know which needs they hope to satisfy, while acknowledging their sample may be used in many ways.
4. ***Polish the presentation.*** Readers will assess whether a sample is relevant based on extrinsic features like syntax highlighting, author credentials, and the explanations accompanying it.

5. *Simplify the code.* Readers desire sample programs to be simple. Common approaches for simplifying the code include keeping it concise, replacing code with placeholders, and showing the code step-by-step.
6. *Keep the code complete.* Sample programs will be copied wholesale into a reader’s own program, bugs and all. Readers desire sample programs that they can use as starting points for their own programs, and which they can modify and incorporate into their existing code. Sample programs would ideally, therefore, be complete, bug-free, and easy to modify.

The guidelines above resemble those shared within programming communities today, like *The SSCCE* (Thompson 2020), the Stack Overflow guidelines for minimal, reproducible examples (Stack Overflow Help Center 2020), and the Mozilla Developer Network code example guidelines (MDN contributors 2020). This chapter has provided empirical evidence supporting each guideline.

The tools in this dissertation support authors in reducing fragmentation (Guideline 1, Chapter 5), making code simple (Guideline 5, Chapters 4 and 6), and keeping code complete (Guideline 6, Chapters 4 and 5). They do so by providing authors with new and improved ways to select, simplify, supplement, and sequence their existing programs. Authors may find inspiration in the above guidelines. Tool builders looking to support the process of program distillation may wish to help authors with the four tasks of selecting, simplifying, supplementing, and sequencing code, while heeding the guidelines for program presentation above.

Chapter 3. Related work

Tools and techniques for authoring sample programs

For as long as there have been programs to share, programmers have needed effective ways to present those programs. Some of the first media for explaining programs were permanent pen and code comments (Figure 3.1). More recently, tool builders have invested considerable effort in developing code editors and program analysis techniques to help programmers present their code to others.

This chapter reviews these tools. First, it reviews what is known about the design and usability of tools for authoring sample programs (page 21). Second, it surveys a broader set of interaction and program analysis techniques that may one day make useful additions to distillation tools (page 43). Finally, it concludes with a proposal of a design space of distillation tools (page 51). This design space maps out promising design opportunities for future tool designers.

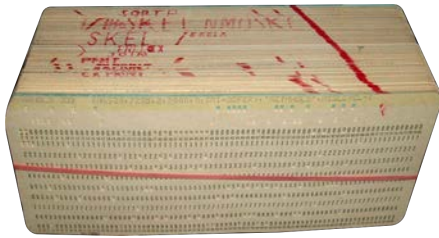
Tools for authoring sample programs

This section reviews tools that support the authoring of sample programs. Three types of tools are reviewed: automated sample generation (page 21), literate programming (page 31), and multi-stage sample authoring (page 40). Each subsection ends with a discussion of how the tools relate to those proposed in this dissertation.

Automated generation of sample programs

What if sample programs could be generated automatically, without any author effort? Such techniques could not only save authors the time it takes to produce illustrative samples, they could also maintain the samples as their dependencies change. This question has led to a rich line of research which crosses software engineering and human-computer interaction.

This subsection describes three ways that automated tools can support the generation of sample programs. First, they can *extract* sample programs from



(a) A deck of punched cards, marked with pen to indicate which cards belong to which procedures (Reinhold 2006).

Programme for Binary to decimal conversion
Add m_6 to all memory locations.

Location	Contents	Remarks
0	$+(23) \rightarrow cA(1)$	Binary number. B
1	$Cc > 0(2) < 0(3)$	
2	$(20) \rightarrow cR(5)$	If B positive 0 sent to 32nd stage of register
3	$-(23) \rightarrow cA(4)$	Absolute value of B obtained
4	$(21) \rightarrow cR(5)$	If B negative 1 sent to 32nd stage of register

(b) An example program from one of the first programming textbooks (Booth and Booth 1956). Each line appears with a description of what that line does.

C ← FOR COMMENT	STATEMENT NUMBER	CONTINUATOR	FORTRAN STATEMENT	IDENTIFICATION
1	2	3	4	72 73 80
			PROGRAM FOR FINDING THE LARGEST VALUE	
C			ATTAINED BY A SET OF NUMBERS	
		X	BIGA = A(1)	
			DO 20 I = 2,N	
			IF (BIGA - A(I)) 10, 20, 20	
	10		BIGA = A(I)	
	20		CONTINUE	

(c) A Fortran program from one of the first manuals for the language (Backus et al. 1956). Statements marked with “C” are descriptive comments.

Figure 3.1: Classic techniques for presenting programs. Even in the earliest days of programming, programmers found ways to make their programs more readable for themselves and others, from marking up their decks of punched cards to carefully typesetting samples.

existing programs. Second, they can *synthesize* sample programs de novo. Third, they can *collect interesting sample data* from a running program. These tools could support the distillation tasks of selecting and simplifying code, and supplementing it with sample data. A comparison of the tools is shown in Table 3.1.

Extracting samples from existing programs

One way to generate sample programs is by extracting them from existing programs. The main idea behind this approach is that for many of the samples an author might create, there already exists a program that can be simplified into that sample. The role of a tool for sample program extraction is to detect important programming patterns in existing programs, extract code from those programs, and format it into a readable and reusable sample.

The tools that researchers have developed for extracting samples all use approx-

	one program	many programs	constraints	sample code	executable	sample data	interactivity	integration with docs	slice	dynamic analysis	symbolic analysis	program execution	program synthesis	remove code	merge code	insert blocks	rename placeholders	rename variables
	Input			Output				Analysis				Edits						
CEG		✓			✓						✓							
Prospector	✓	✓	✓	✓		✓					✓							
eXoaDocs	✓		✓				✓	✓				✓	✓					
Buse and Weimer	✓		✓							✓		✓	✓	✓	✓	✓	✓	✓
Codex	✓		✓									✓						
OverCode	✓		✓	✓		✓			✓			✓						✓
CodeHint	✓	✓	✓	✓		✓			✓		✓							
MUSE		✓	✓				✓	✓				✓	✓					
SpyREST	✓				✓		✓	✓										

Table 3.1: A comparison of tools for automated sample program generation. These tools generate sample programs, and sample data for programs to operate on. Some tools *select* code (see “slice” column). Others automatically *simplify* extracted programs to make them more readable and reusable (see “Simplifications” column family). Yet other tools *supplement* documentation for a program with sample data (see “sample data” column).

imately the same process. First, the tool scans a collection of existing programs. When it finds an interesting pattern in a program, it extracts the fragment of the program containing the pattern. Then, the tool picks representative fragments to create samples from. To pick representative fragments, the tool clusters the fragments, ranks them according to some measurement of readability or reusability, and picks the top-ranked fragments. The chosen fragments are simplified with automated program transformations, and then shown to the user.

Let’s consider an example. eXoaDocs (Kim et al. 2010) was one of the first published systems for sample program extraction. Given the name of a method from an API, eXoaDocs could generate a usage sample for that method. It generated samples by querying the Kodex code search engine for uses of the API method. For each use that was found, the tool extracted a “slice”¹ of code leading up to that method use. It clustered the slices using a set of automatically-computed semantic features. Samples were chosen by ranking slices according to their conciseness and correctness and then showing the user the top-ranked slices (Figure 3.2). Apart from the slice operation, the sample program was not further simplified.

¹ See the definition of slicing in the subsection on efficient program selection (page 44).

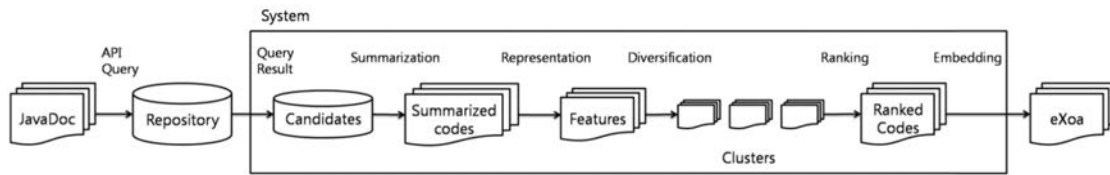


Figure 3.2: A workflow for extracting sample programs from existing programs. This figure for eXoaDocs (© 2009 IEEE, Kim et al. 2009) shows how it generates samples by mining programs from a repository, selecting relevant code from the programs (i.e., “summarizing”), clustering them, and returning representative samples from those clusters.

Many other sample program extraction tools have been developed. These tools can generate several types of samples, and use several techniques for selecting code from existing programs and simplifying samples.

What types of samples can be extracted? Tools can generate samples that are either *concrete* or *abstract*. *Concrete* samples are copies of real, existing programs that may have been lightly modified (Kim et al. 2010; Montandon et al. 2013; Moreno et al. 2015). *Abstract* samples are samples that are generated from scratch to represent an entire cluster of program fragments (Buse and Weimer 2012; Fast et al. 2014; Allamanis and Sutton 2014; Glassman et al. 2015).²

Some tools produce samples that can be *compiled and executed*, and others do not. In general, it is difficult to extract executable samples if the existing program is sufficiently complex. The tool might not know whether it managed to extract all of the code necessary for the sample to run. Some of the tools use slicers to extract code (see Kim et al. 2010; Montandon et al. 2013; Moreno et al. 2015). These slicers may only be able to extract code from a single method, and sometimes miss program dependencies that can’t be inferred from the code. That said, some tools do generate executable samples. For example, OverCode (Glassman et al. 2015) constructs canonical solutions to programming problems by clustering student submissions. It keeps samples executable by not removing any code from the student submissions that it turns into samples.

Automated code selection. Tools can extract code for sample programs by slicing existing programs. For instance, tools for generating API usage samples extract slices containing a usage of an API member, along with a set of statements that the usage depends on (Kim et al. 2010; Montandon et al. 2013; Moreno et al. 2015). If a tool generates abstract samples, it can use a statistical approach to determine

² These definitions are based on Moreno et al.’s (2015) definitions. In this chapter, the terms refer to samples of all types, and not just API usage samples.

```
FileReader f; //initialized previously
BufferedReader br = new BufferedReader(f);
while(br.ready()) {
    String line = br.readLine();
    //do something with line
}
br.close();
```

Figure 3.3: An automatically-generated sample program. This sample was generated using Buse and Weimer’s (2012) tool. While the sample is based on an existing program, several parts of that program have been simplified. Statements have been replaced with placeholders (“//do something with line”). The variables have been automatically renamed. Code listing reused with permission (© 2012 IEEE, Buse and Weimer 2012).

what statements should appear in the sample, including only the statements that appear in a large number of existing programs (Buse and Weimer 2012).

Automated sample simplification. To simplify a program, a tool can either apply automated transformations, or return what it infers to be the simplest fragments extracted from existing programs. Buse and Weimer’s (2012) tool provides what may be the most advanced automated transformations for simplifying programs. Given a cluster of program fragments, the tool constructs abstract samples by selecting statements that are shared across the program fragments. If this yields a sample program with two adjacent control blocks with the same guard (e.g., two `if` statements with the same condition), those blocks are merged. Uses of variables that are not of the type for which a sample was requested get replaced with generic placeholders like “//do something with line.” (Figure 3.3).

Tools that create abstract samples can assign idiomatic names to variables by observing which names are the most common within the cluster the sample is being created from (Buse and Weimer 2012; Glassman et al. 2015). Variable names can also be generated according to simple typographical rules, such as abbreviating the name of a variable type (Buse and Weimer 2012).

If a tool generates concrete samples, its primary mechanism for simplifying the programs is to show the users the simplest fragments from each cluster. For instance, MUSE (Moreno et al. 2015) incorporates three types of information about a sample when ranking samples to show to a user. First, the sample’s popularity, measured by the number of times that the code for that sample appears more or less duplicated in existing programs. Second, its readability, estimated using an automated metric of source code readability (Buse and Weimer 2010) well-known within the software engineering research community. Third, its reusability, measured in a way that penalizes samples that make use of unusual data types.

How usable are the extracted samples? For many of these tools, studies have affirmed that when programmers have access to samples generated by the tools, they complete programming tasks more efficiently (Kim et al. 2010; Moreno et al. 2015; Glassman et al. 2015). Two studies in particular indicate how the tools compare to each other and to the work of human authors.

Buse and Weimer compared the samples their tool generated with those generated by eXoaDocs, and those written by human authors in the Java documentation. Using an automated program readability assessment tool they had published previously (Buse and Weimer 2010), their samples were scored as more readable than human-written samples and those from eXoaDocs. When asked to judge the quality of samples in head-to-head comparisons, students in a software engineering course judged Buse and Weimer’s samples to be as good or better than those by human authors 60% of the time, and strictly better than those from eXoaDocs 75% of the time. However, graduate student respondents seemed to have a reduced preference for the tool-generated samples when compared to undergraduate students. These results indicate that generated samples may under certain circumstances approach the quality of human-generated samples.

Moreno et al. (2015) conducted three studies of samples produced by MUSE. One study showed that the ranks MUSE assigned to samples correlated with 9 developers’ ratings of the usefulness of those samples. In a second study, 119 developers rated the usefulness of the generated samples and provided qualitative feedback on them. 82% of samples were rated as either “very useful” or “useful.” The most frequent reason for negative scores was the unnecessary complexity of the samples. For instance, 38 of the 44 lines in one sample were seen as irrelevant, though were included because the program slicer saw them as relevant. Respondents also wanted to see the result of running the code, and to view alternative usage scenarios for a method. This feedback indicates how future innovation in sample generation tools may yield better samples for readers.

Synthesizing sample programs

Are existing programs necessary for generating sample programs? Or can sample programs be generated from nothing but a specification?

The task of generating programs that satisfy a user intent is called program synthesis (Gulwani et al. 2017). Given a small set of constraints specified by a user, a synthesizer produces a program that matches those constraints by efficiently searching a large space of potential programs in an underlying language. Sample program generation could be considered a special case of program synthesis, where the output is meant to be read and reused by a programmer.

Rissland and Soloway (1980) introduced one of the first tools for sample generation by synthesis. They called their approach constrained example generation (CEG). Their goal was to generate LISP data structures for exercises in an intelligent tutoring system. The tutoring system requested lists with specifications like, “Give an example of a list of 0’s and 1’s, which is longer than 2, and which has at least one element deeper than depth 1.” The tool then searched through a space of possible LISP lists, repeatedly modifying a set of simple lists by adding elements to them, and increasing the depth of elements. The system returned the first generated list that matched the specification.

Synthesizing samples in general-purpose programming languages. Several tools support the synthesis of samples in general-purpose, imperative programming languages. Prospector (Mandelin et al. 2005), for instance, synthesizes Java snippets. A user specifies a data type that they want a synthesized snippet to produce, and the types of objects they currently have in their program that can be used as inputs. To synthesize a snippet, Prospector first constructs a graph, where each node is a data type, and each edge is an operation (e.g., a method) that maps from types it can take as input to the types it produces upon execution. Snippets are synthesized by finding a path through the graph, or a sequence of operations, that connect the input type to the output type.

CodeHint (Galenson et al. 2014) provides additional flexibility for how users specify the sample they want. A user invokes CodeHint when they are writing a program of their own and want a snippet that produces a specific value. The user can provide a specification for a snippet in the form of a desired output, dynamic type constraints, program sketches, or some combination of all of these. CodeHint inspects the variables in the running program’s memory to infer what expressions can be used to transform the variables into the desired output. It iteratively constructs candidate programs by composing these expressions. Each candidate program is tested against the specification by executing it, and the program state is rolled back whenever a tested programs causes undesired side effects.

The use of existing programs in synthesis. Technically, neither Prospector nor CodeHint needs to refer to existing programs to generate samples. That said, both of these tools benefit from rules learned from existing programs. Prospector learns how to cast some API types to other types from existing samples, and CodeHint learns whether it can use specific static constants as an arguments to specific methods. These enhancements serve as reminders that a tool may be able to better produce programs by taking note of what a typical program looks like.

Synthesizing readable programs. While studies have shown that the systems above generate samples that are useful to programmers (Mandelin et al. 2005; Galenson

et al. 2014), there have not yet been studies comparing the quality of the generated samples to those produced by human authors. Synthesizing understandable, modifiable programs is a challenge for program synthesis and its companion discipline in human-computer interaction, programming by demonstration (Cypher 1993). The challenge of making synthesized programs easy to understand has been recognized within the programming by demonstration community for some time (Myers et al. 2000; Lau 2009).

Program synthesizers are thus sometimes built with readability in mind, especially when users are expected to modify the synthesized program (Mayer et al. 2015; Chasins et al. 2018; Hempel et al. 2019; Drosos et al. 2020). To synthesize a readable program, a synthesizer might prioritize candidate programs that are short and consist of common programming idioms (Gulwani et al. 2017, p. 54). One notable example is the CoScripter system (Leshed et al. 2008), which generates programs from demonstrations in pseudo-natural language text, which can later be modified by end users. For these tools, empirical studies of readability like those summarized in Chapter 2 may be necessary to confirm which design decisions for synthesizers lead to more readable programs.

Collecting sample data from running programs

Programs are often documented with *sample data*, or inputs the program can be invoked with, paired with the output the program produces when run on those inputs. Readers benefit from sample data as they can use the sample inputs to test out the program when they are using it for the first time, and view the outputs to understand what exactly the program does.

Tools can help authors generate sample data by recording the inputs submitted to a program and outputs produced by the program as it runs, and then packaging this information into documentation. SpyREST (Sohan et al. 2015), Meta (Fast and Bernstein 2016), Vesta (Krämer et al. 2016), and DynamiDoc (Sulír and Porubán 2017) all generate sample data in this way, for REST API endpoints, Python functions, JavaScript functions, and Java functions, respectively. SpyREST, for instance, generates sample data for REST APIs by intercepting client requests to APIs, and then logging the request body, the name of the endpoint, the version of the API, and the response that the API returned. This data is then cleaned up, formatted, and inserted into documentation.

These tools could potentially help authors produce more, and better, sample data. As Krämer et al. (2016) note, a programmer executes their program on sample inputs as they write it, but cannot be bothered to stop programming to document these inputs. As a result, sample data is often lost. The above tools

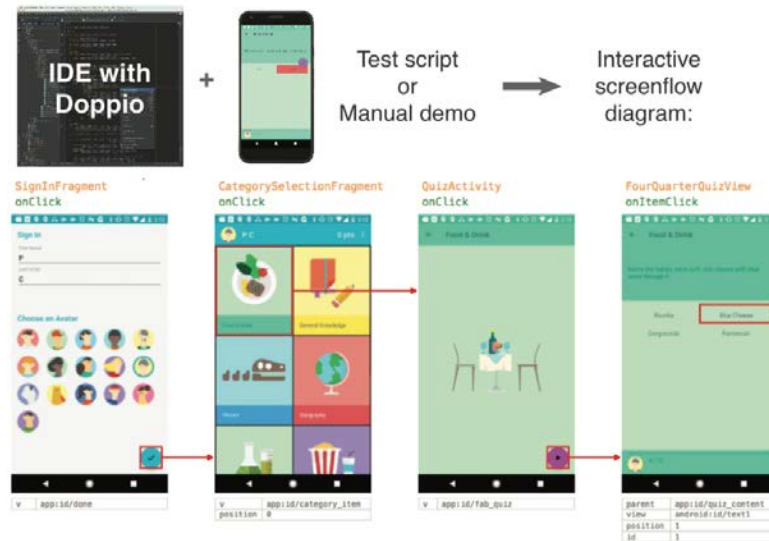


Figure 3.4: A sample usage of a mobile app, captured from author demonstration and rendered as a mixed-media flow diagram. Video is captured as an author uses the app. A recording of the video is segmented, and annotated with transitions linking button presses to screen changes, using data collected from an instrumented version of the app. This figure first appeared in *Doppio: Tracking UI Flows and Code Changes for App Development* (Chi et al. 2018), published under the [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/).

ensure they are retained. And indeed, during an in-lab study, programmers who used Vesta had more complete and accurate documentation at the end of the study than those who did not. Furthermore, if tools collect sample data as the programs are deployed to real users, as is the case for SpyREST and Meta, they can collect sample data from these real users, rather than from the author. Such data could represent real world use cases better than an author’s sample data.

To make sample data more readable, tools can transform it. SpyREST simplifies sample data, truncating API responses that it deems to be too long. Both SpyREST and Vesta generate test code that can be used either by readers to try out the program, or by authors to test their code. DynamiDoc generates string descriptions of function behaviors like “When `ch = 'A'`, the method returned `'\u0041'`.” And both DynamiDoc and Meta, when many sample inputs and outputs have been logged, prioritize which ones to show to readers based on which inputs the program was invoked with most frequently.

If a program is interactive, tools can capture and document user experiences. Doppio (Chi et al. 2018), for example, creates mixed-media diagrams of Android app usage (Figure 3.4). The diagrams consist of embedded video clips of each screen, in order, that an author opened as they used an app. Arrows link one screen to another, pointing from a button that was clicked on one screen to the

other screen that was opened as a result. As the designers of this tool found in in-lab studies, rich sample usages like these can help readers understand programs (Chi et al. 2018). That said, generating such rich representations of usages requires sophisticated program instrumentation. Doppio records video as the author uses the app, and logs which functions were invoked and when, so that it knows how to segment screen recordings and link them in the flow diagram.

Tools for collecting sample data can serve yet another purpose in program distillation: they can help authors *simplify* sample programs. As will be discussed in Chapter 4, tools can collect the values that variables take on as a program executes. They can then give authors the choice to replace a reference to a variable with its value. This lets an author eliminate the code used to produce the value, and thus shorten the sample program.

The role of authors in sample program generation

A few exceptions aside, the tools described in this subsection do not allow an author to influence the design of the sample program. Tools that generate sample data from runtime instrumentation (Sohan et al. 2015; Fast and Bernstein 2016; Krämer et al. 2016; Sulír and Porubán 2017) require authors to use the program, and tools for generating sample programs by synthesis (Mandelin et al. 2005) accept a specification of program behavior from users. However, generally, these tools do not allow an author to express how they wish the program to *appear*.

Three tools from this subsection deserve mention for the agency they afford authors in influencing the generated sample. CoScripter synthesizes programs as executable pseudo-natural language instructions and lets authors directly edit these instructions (Little et al. 2007). CodeHint lets authors write only the parts of the program they care to specify, and leave the rest to the synthesizer to generate (Galenson et al. 2014), an approach known as program synthesis by sketching (Solar-Lezama 2008). Doppio generates flow diagrams showing how one might use an app, taking an author demonstration as input (Chi et al. 2018). These tools combine the power of automated program analysis and synthesis with author input that specifies what the sample should do and how it should appear.

Limitations of sample program generation tools

The tools in this subsection have three limitations. First, extracted samples are often either incomplete or larger than they need to be, due to imprecision in program slicers. Second, the tools do not permit authors to influence the design of the samples, as noted above. These first two limitations are addressed in Chapter 4,

which introduces an interactive tool that lets authors work together with a slicer to extract complete, concise snippets. This tool also lets authors provide input into how they would like to simplify the sample.

The third limitation is that these tools have not yet been shown to be capable of generating larger samples like programming tutorials. In Chapter 6, an interactive tool is presented that can help authors create tutorials, and author needs are identified that future automated tools may be able to assist with.

Literate programming

In 1984, Donald Knuth proposed *literate programming* as a new approach to writing code. In this vision, instead of programs, authors write about computational ideas and the implementation of those ideas. Instead of simply commenting their source code, a programmer splits their program into brief code snippets, and interleaves these snippets with explanations about what the snippets do, and how they fit together into a complete program. The output of literate programming is a document that describes an algorithm, studded with code that shows how each piece of the algorithm is implemented (Knuth 1984).

Those familiar with the specifics of literate programming will note that the term is used liberally in this subsection, compared to what Knuth might have considered a literate programming tool. The common thread connecting all of these tools is the belief that programs should be written and distributed in a medium where code is accompanied by explanations. A comparison of the tools discussed in this subsection is shown in Table 3.2.

The beginning: WEB and related tools

In his seminal article on literate programming, Knuth (1984) presented a tool called WEB. A programmer would write a program in a WEB file, which would comprise many sections, each containing commentary about the program, and optionally, a program snippet. WEB's two processors transformed the file into more useful representations. The TANGLE processor combined program snippets into a complete, machine-executable program. The WEAVE processor formatted the WEB file into a nicely typeset document, complete with cross-references between the definitions of program snippets and where they were used (Figure 3.5).

Most unique to WEB was the ability to order code flexibly, in a way that was not possible with a typical compiler. Snippets could be shown in practically any order an author might wish, because they were defined in terms of other snippets,

	script	"web"	notebook	project	revision history	code	rich text	output	cross references	monolithic program	cell-by-cell	reactive	cache partial results	show / hide snippets	guided tour	execute code	interactive dashboard
	Program Format			Contents				Execution				Reader UX					
WEB	✓				✓	✓	✓	✓	✓				✓				
Jupyter notebook		✓			✓	✓	✓			✓	◆		✓	✓	◆	◆	
JTourBus			✓		✓								✓				
Tempe		✓			✓	✓	✓			✓			✓	✓			
Codepourri	✓				✓	✓							✓	✓			
Observable		✓			✓	✓	✓			✓			✓	✓	✓		
chat.codes			✓	✓	✓	✓		✓					✓				✓
Streamlit	✓				✓	✓	✓			✓	✓		✓	✓	✓		

Table 3.2: A comparison of tools for literate programming. Tools for literate programming help authors *supplement* programs with rich text and outputs. They also support authors in *sequencing* code in new ways, i.e., as webs of snippets (Knuth 1984). The diamond sign (◆) indicates that a tool provides the functionality only with community-provided extensions.

and TANGLE figured out how to assemble working programs from snippets based on references between them. Two segments of the same function, then, could appear on two different pages. A program could be presented using a bottom-up approach (i.e., showing the algorithmic details first), a top-down approach (i.e., leading in with the program structure), or some combination of the two approaches.³

Literate programming attracted great interest within the computer science community for some time. Knuth published literate programs in the *Programming Pearls* column of the *Communications of the ACM*, and there was even a literate programming column in the *Communications* (Van Wyk 1990). Childs (1992) estimated there were at least 1,000 users of literate programming tools at one point. A series of other literate programming tools, among them CWEB (Thimbleby 1986) for C, and Noweb (Ramsey 1994) for any language, were developed.

Knuth claimed that WEB programs could be written as quickly as those in other languages, and that WEB programs were not only better documented, but also easier to understand and debug. Though just how usable were WEB-like tools?

One early criticism of literate programming was that it was inadequate for everyday engineering work, where simple systems built quickly from reliable existing

³ The reader is encouraged to look at an even more complex WEB program in the [preface](#).

```

@ This program has no input, because
we want to keep it rather simple.
The result of the program will be to
produce a list of the first thousand
prime numbers, and this list will
appear on the |output| file.

...

@<Program to print...@>=
program print_primes(output);
const @!m=1000;
@<Other constants of the program@>;
var @<Variables of the program@>;
begin @<Print the first |m| prime
      numbers@>;
end.

```

WEB code

```

2. This program has no input, because we want to
keep it rather simple. The result of the program will
be to produce a list of the first thousand prime
numbers, and this list will appear on the output file.

...

<Program to print the first thousand prime
  numbers 2 > ≡
program print_primes(output);
  const m = 1000;
  < Other constants of the program 5 >
  var < Variables of the program 4 >
  begin < Print the first m prime numbers 3 >
  end.

This code is used in section 1.

```

generated document

Figure 3.5: A section of a WEB program and the document generated from it. When writing a WEB program, authors write code as sections, each containing a Pascal code snippet and a prose description of the snippet. Snippets are built from snippets defined in other sections. Cross-references between each snippet’s definition and uses are automatically inserted into the generated document. The excerpts in this figure are reused from (Knuth, *Literate Programming, The Computer Journal*, 1984, Volume 27, Issue 2, pages 97–111) with permission from Oxford University Press.

programs were better than well-explained, meticulously-designed novel systems. In such contexts, WEB programs were “a sort of industrial-strength Faberge egg” (Bentley et al. 1986).⁴ It was also implied that the needs for literate programming were specialized enough that any practitioner of literate programming had written their own WEB-like tool (Van Wyk 1990).

Observations of the tools in use provide some evidence of their usability. Ramsey and Marceau (1991) described their own use of WEB in developing a 33,000-line software system on a team of seven. Among their takeaways were that the tool should preserve the formatting of the original code (a feature introduced in both CWEB and Noweb) and support the inclusion of diagrams and pictures. The team found themselves in need of generating tables of contents with many levels of hierarchy, given the size of the code. They also wished that the team’s own familiar tools could be used for formatting text (in their case L^AT_EX rather than T_EX). These suggestions aside, the team felt that the juxtaposition of design documentation with code eased the effort of maintaining the code, and that the documentation for this system was used more often than that for other systems the team had built, because it was so close to the source code.

⁴ Knuth himself noted that WEB might only be for “the subset of computer scientists who like to write and to explain what they are doing” (1984).

Two studies of the use of literate programming in the classroom provided further evidence of the tools' uses and usefulness. Childs et al. (1995) taught WEB to an introductory programming course of honors students, reporting that these students, compared to those taking prior offerings of the course, performed better in a later data structures course. Shum and Cook (1994) conducted an experiment where students in a junior computer science programming course completed assignments using either a literate programming tool or a comparison tool, Turbo C. Students using the literate programming tool wrote significantly more documentation, even though they wrote essentially the same number of lines of code. They also wrote comments about algorithm design and examples of what the code did, while no such comments were written by students in the control condition. This latter study suggests that, in line with Knuth's goals, literate programming tools may lead programmers to write better explanations of their programs.

The tools just described reveal enthusiasm in the computer science community for tools that enable careful program presentation. While the WEB family of tools does not seem to be used widely today, the idea of writing code alongside documentation lives on in many tools, among them computational notebooks.

Computational notebooks

In recent years, computational notebooks have become a popular type of program editor. A notebook represents an interactive computing session, rather than a monolithic program. In a notebook, a programmer writes snippets of code inside *cells*, and can submit each of these cells one at a time to be executed by an interpreter. The interpreter returns outputs, which are embedded in the notebook next to the code that produced it. Programmers can order cells however they see fit. They can add rich text descriptions between cells to record their own notes, interpretations of results, and documentation (Figure 3.6). Because notebooks are focused on supporting interactive computing, the creators of Jupyter notebook have called them environments for *literate computing* (Perez and Granger 2015).

Notebooks today have millions of users. They have been created for languages including Python (Jupyter), JavaScript (Observable), and R (Knitr). Notebook-like tools have even been developed for command-line programming environments (Schulte et al. 2012). While these tools are designed to support the explanation of programs, recent studies have shown that sometimes users face considerable challenges using notebooks to present their code.

Messes in computational notebooks. Even though notebooks provide features for explaining code, users of notebooks often find it challenging to prepare the program in their notebook for an audience. This is because their code was written for their

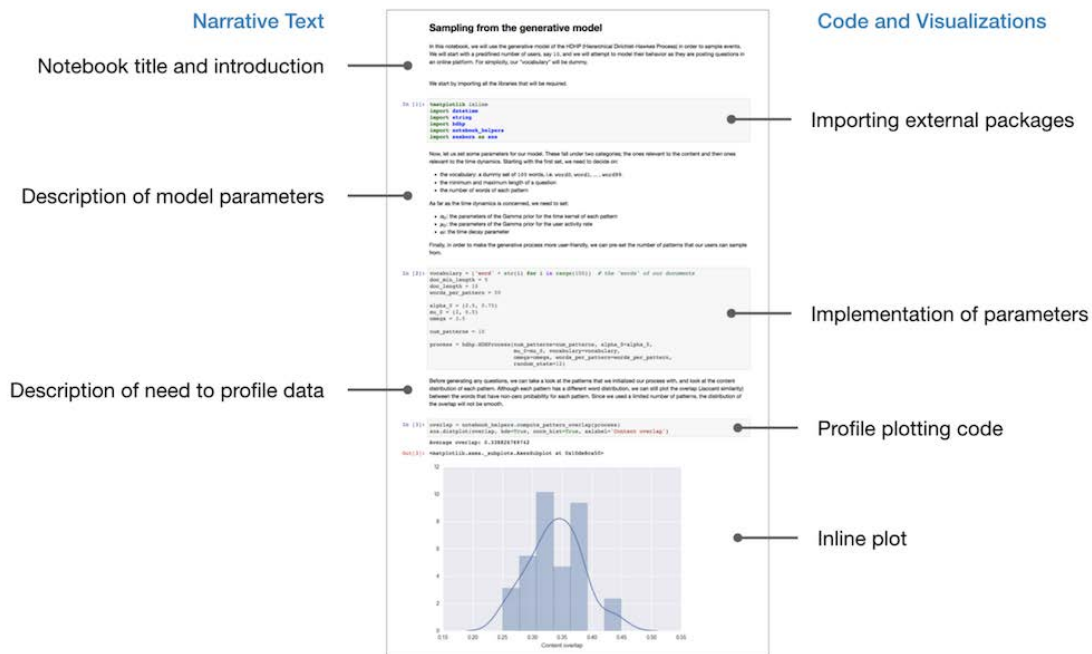


Figure 3.6: A schematic of a computational notebook. This is a Jupyter notebook, containing interspersed cells with text descriptions, code for data analysis, and outputs of running the code. This figure is reused from Rule’s dissertation *Design and Use of Computational Notebooks* (Rule 2018), published under the Creative Commons Attribution 4.0 International License.

own exploration, and now they must repurpose it for explanation (Rule et al. 2018c). Programmers have called their notebooks “messy” (Kery et al. 2018; Rule et al. 2018c), containing “ugly code” and “dirty tricks” in need of “cleaning” and “polishing” (Rule et al. 2018c).

Messes in computational notebooks can be attributed in part to the lackluster code quality that appears to be intrinsic to exploratory programming. Programmers regularly prioritize the efficient discovery of solutions over writing high-quality code (Kery and Myers 2017). They clutter their programs by saving old versions of their code in comments (Yoon and Myers 2012; Kery et al. 2017). In notebooks in particular, poor code quality takes on a spatial dimension. Messes accrue and disappear in an iterative process of expansion and reduction of code: programmers write code in many small cells to try out different approaches to solve a problem, view output from their code, and debug their code; and then combine and eliminate cells as their analyses reach completion (Kery et al. 2018).

Eventually, messes start getting in a programmer’s way. It becomes difficult to understand analyses split across many cells of a notebook, and long notebooks

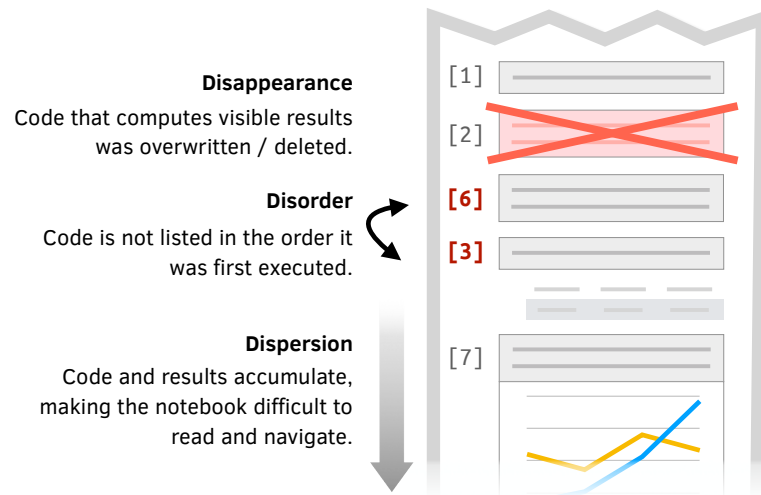


Figure 3.7: Types of messes in computational notebooks. These messes make it difficult for a programmer, and other readers of the notebook, to find results of interest and the code that produced them.

become time-consuming to navigate (Kery et al. 2018). Important results accidentally get overwritten or deleted (Rule et al. 2018c). While programmers often wish to share their findings with others (Kandel et al. 2012; Kery et al. 2018; Rule et al. 2018c), they are often reluctant to do so until they have cleaned their code (Rule et al. 2018c; Rule et al. 2018a) (Figure 3.7).

To manage these messes, programmers have adopted diverse strategies to clean their code. Many delete cells they no longer need, consolidate smaller cells into larger cells, and delete full analyses that did not turn out helpful. Long notebooks are abandoned for “fresh” ones with only a subset of successful parts from the long ones. Analysts organize code as they build it, some coding from top to bottom, some adding cells where they extend old analyses, some placing functions at the top, and some placing them at the bottom (Kery et al. 2018). They add tables of contents, assign numbers to sections, limit the size of cells, and split long notebooks into shorter ones (Rule et al. 2018c).

Because of the challenges and tedium of managing messes, programmers have clearly indicated that they need better tools to support the management of messes. In prior studies, programmers have asked for tools that let them collect scripts that can reproduce specific results, compare outcomes from different versions of an analysis, recover copies of notebooks that produce a version of a result (Kery et al. 2018), and recall the history of how data was created, used, and modified (Rule et al. 2018a). Answering this call, recent tools have provided capabilities for programmers to save and navigate versions of code and results (Kery and Myers 2018; Rule et al. 2018a; Kery et al. 2019) and to collect “recipes” of notebook code

that reproduce selected results using program slicing (Kery and Myers 2018).

Making notebooks clean by design. Could notebooks be redesigned so they don't get messy? Tool builders have designed several extensions to the notebook to help programmers manage messes. One extension allows authors to hide messes by collapsing snippets and outputs they do not wish to see. This feature has been prototyped in research tools (DeLine et al. 2015; Rule et al. 2018a) and incorporated into widely-used notebooks (Jupyter, Observable). Taking this idea to an extreme, a tool could hide *all* code by default, and only show readers the code the author explicitly asks to be shown. This is the approach taken by Streamlit, an authoring tool for notebook-like data apps. Authors write programs as scripts, from which Streamlit generates a dashboard of the program's results and control widgets for exploring those results. If the author wishes for the reader to see any block of code, they must wrap that code in a special function.

A notebook can also prevent messes by stopping the author from accumulating old code and results. One way to do this is to run cells *reactively*. Observable, for instance, infers dependencies between cells. A dependency is found whenever one cell uses a variable defined in another cell. When an author executes a cell, the cells that depend on it are executed afterwards, ensuring that all outputs will reflect the most recently executed version of each cell. A notebook can also update outputs *live*. Tempe, a notebook for analyzing streaming data (DeLine et al. 2015), listens for edits to cells. As soon as an author finishes editing a line, Tempe re-executes the edited line, and all other lines that depend on it.

Which model of execution is better: executing one cell at a time, or updating outputs reactively and live? To answer this question, DeLine and Fisher (2015) conducted a controlled experiment in which data-experienced professional programmers used two versions of Tempe. One version consisted of a read-eval-print loop (REPL) interface to an interpreter like the RStudio console. This version provided an append-only history; unlike Jupyter notebook, participants could not arrange, edit, or delete cells. The other version was a live, reactive notebook where a user's edits triggered re-execution of the code and updated the results.

DeLine and Fisher found that when programmers used these two versions of the tool for exploratory data analysis, the histories left by both tools contained the same number of results that participants marked as "insightful." Those created with the live tool had less errors. Participants preferred the live version according to 11 dimensions of usability, except when it came to ease of learning and ease of use. Participants, it was noted, may have had more prior experience with REPLs. Participants appreciated the responsiveness of the live tool and its ability to keep the script content clean. That said, the REPL had the advantage of preserving a complete log of the code executed and the results produced.

DeLine and Fisher’s study shows that the execution model of a notebook can affect a programmer’s perception of the cleanliness of their code and their own programming effectiveness. Of course, the design space of notebooks is much broader than the two designs tested by DeLine and Fisher. How much of the benefit of live computation comes from re-executing a cell after a user’s edits, and how much comes from reactive computation? Should programmers be allowed to rearrange cells, and if so, should the movement of cells be constrained by the tool so that uses of variables always appear after their definitions?

Authoring tools for guided program tours

Could a programmer create a document with interspersed code and explanations by annotating programs that already existed, instead of writing the program from scratch in with a computational notebook or in a WEB file? One such format is the *program tour*. Tours guide a reader to see one location in a program after another, with explanations of the code at each of those locations. Tours can quickly show a reader points of interest in even massive programs.

Creating a tour. To create a tour, an author needs to create steps, associate code with those steps, and explain each step. A tool can let authors do this *directly*, by selecting and annotating code within a code editor or browser (Suzuki 2015; Oney et al. 2018). Other tools require authors to *annotate the source program* by, for instance, adding special comments to indicate what code should show in the step, and when in the sequence of steps that code should be visited. For other tools, authors create an entirely new metadata file, listing the explanations for step, and providing a selector that will be used to detect the code to be highlighted in that step (Google 2020[a]) (see, for example, Figure 3.8).

Usability. Studies of program tours have yet to provide evidence of their effectiveness in helping readers navigate and understand code. A study of the JTourBus system showed that readers could finish comprehension tasks more quickly, though not more accurately, when using the tool (Ozbek and Prechelt 2007). That said, tours have been developed and used in the training materials at software development companies like Google (Johnson and Senges 2010).

Crowdsourcing tours. Could tours of programs be crowdsourced? Gordon and Guo (2015) describe the iterative design of Codepourri, a tool for creating tutorials of small programs from line-by-line explanations crowdsourced from learners. Learners were routed to lines and asked to describe what the code was doing at that line. An expert panel found that 65% of the explanations of lines were correct, and 20% were of exceptional quality. The generated tutorials as a whole were of

A doc/codewalk/uripoll.go

```

"http://golang.org/",
"http://blog.golang.org/",
}

```

B // State represents the last-known state of a URL.
type State struct {
 url string
 status string
}

```

// StateMonitor maintains a map that stores the state of the URLs to
// be polled, and prints the current state every updateInterval nanosec.
// It returns a chan State to which resource state should be sent.
func StateMonitor(updateInterval time.Duration) chan<- State {
  updates := make(chan State)
  urlStatus := make(map[string]string)
  ticker := time.NewTicker(updateInterval)
  go func() {
    for {
      select {
      case <-ticker.C:
        logState(urlStatus)
      case s := <-updates:
        urlStatus[s.url] = s.status
      }
    }
  }()
}

```

C **State type**
 The State type represents the state of a URL.
 The Pollers send State values to the StateMonitor, which maintains a map of the current state of each URL.
 doc/codewalk/uripoll.go:26,30

D <step title="State type" src="doc/codewalk/uripoll.go:/State/,/}>
 The State type represents the state of a URL.

Resource type
 A Resource represents the state of a URL to be polled: the

Figure 3.8: A guided tour of a program. This Codewalk provides a guided tour of a sample program (A). Throughout the tour, sections of code are highlighted (B) along with prose explanations of that code (C). Codewalks are written as XML specifications, where each step is specified as a selector determining what code should be highlighted, and a prose explanation (D). This figure is a modification of the Codewalk *Google 2020(b)* from the Go language documentation. The documentation is licensed under the Creative Commons Attribution 3.0 International License and the code is licensed under a BSD license.

comparable quality to the ones the experts had written themselves. Gordon and Guo proposed two methods for incorporating learner feedback in filtering explanations to just those that were the most helpful. While the system was only tested on short programs, this study suggests that it might be possible to create tours of some programs without the input of its original author.

Limitations of literate programming tools

This dissertation addresses two limitations of literate programming tools. The first limitation is the messiness of computational notebooks. To complement recent research systems that help programmers forage for code (Kery and Myers 2018; Rule et al. 2018a; Kery et al. 2019) and extract “recipes” of code that produced results in notebooks (Kery and Myers 2018), Chapter 5 contributes the design of a distillation tool for cleaning messy notebooks.

The second limitation is that contemporary notebooks (Jupyter, Observable, DeLine et al. 2015) do not offer flexibility to organize code in ways frequently seen in programming tutorials. For instance, an author cannot split the lines of a function into multiple cells. Chapter 6 introduces a new type of notebook that

	real-time recording	label versions	provide all versions	add commentary	narrate	continuous recording	retroactive editing	cross-version editing	format diffs	multi-stage sample	tutorial	screencast	live demo	scrub over history	scrollytelling	syncd commentary
	Author Input			Tool Support				Output								
JTutor		✓	✓			✓			✓							
Storyteller	✓		✓		✓	✓	✓	✓			✓		✓		✓	
Ginosar et al.		✓				✓	✓	✓	✓					✓		
Torta	✓		✓	✓	✓	✓		✓		✓	✓					
Improv		✓	✓			✓						✓				
Waves			✓			✓		✓		✓				✓		

Table 3.3: A comparison of tools for authoring multi-stage samples. These tools support authors in showing how a sample program is built up step-by-step, either as a sequence of discrete stages, or as a continuous recording of program evolution. They differ in terms of the type of support they provide to help authors *sequence* their program into stages (see “Tool Support” column family).

provides the flexibility of a WEB-based tool, the rich text editing capabilities of a typical notebook, and the live computation of [Observable](#).

Multi-stage sample authoring

Sometimes, authors want to show how a complex program is built over the course of many small additions and changes. The tools in this subsection help authors record, edit, and present the evolution of a program as *multi-stage samples*. The tools help authors create tutorials, screencasts, and live demos, each of which show code as it evolves over multiple stages.

One key distinction among the tools is whether they show the evolution of code as *discrete stages* or *continuously*. Some tools cross this boundary, recording the continuous evolution of a program, and then presenting it as a series of discrete stages. A comparison of the tools discussed is shown in Table 3.3. Below, the four main features of multi-stage sample authoring tools are described.

Step 1. Record program evolution. First, the authoring tool records how the program evolves. Perhaps the most non-intrusive way to do this is to listen to edit events from a programmer’s code editor (Mahoney 2018; Oney et al. 2018), or file save events from the operating system (Mysore and Guo 2017). The tool can require authors to manually tag a version of the code when it has reached a stable stage (Kojouharov et al. 2004; Ginosar et al. 2013; Chen and Guo 2019). Tools can also

require the author to supply versions of the sample program themselves, with one for each stage to be shown to the reader (Pombo 2020).

Step 2. Revise histories. After an author has created several discrete stages, or recorded the program evolution for some time, they may wish to revise this history. Authors have myriad reasons for wanting to do so. For instance, they may want to rewrite a line of code in the first stage it appears in order to fix a bug, or because they have thought of a pedagogically better way of writing it (Ginosar et al. 2013). Tools can help authors propagate an edit to the code backwards in time from the stage they are currently editing (Ginosar et al. 2013; Mahoney 2018). They can also support authors in reordering stages (Kojouharov et al. 2004).

Step 3. Annotate program. If a program is going to be shown without narration or a surrounding text tutorial, an author may want to annotate parts of the program with explanations at its various stages. Storyteller (Mahoney 2018) replays the evolution of a program as a silent screencast. The authoring interface lets authors add comments that will appear in a side bar at a precise edit event. The chat.codes tool (Oney et al. 2018) provides another authoring paradigm, where an author writes explanations of their edits in a chat as they are editing the program. Authors can insert links to selections of code at that moment in history into the chat message by directly selecting the code. Improv (Chen and Guo 2019) lets authors create slides for a live demo consisting of selected program snippets, explanations, and a terminal window where the code can be executed.

Step 4. Present stages. With the knowledge of how a program evolved, the tool can then generate a tutorial, screencast, or some other artifact, optimized for readability and navigability. The output format varies by tool. Storyteller (Mahoney 2018) creates screencasts. Ginosar et al.’s (2013) tool creates a multi-stage sample where each stage can be viewed by dragging a slider. Torta (Mysore and Guo 2017) generates mixed-media tutorials with embedded, cropped screencasts and listings of file changes. JTutor (Kojouharov et al. 2004) creates tutorials that can be replayed within the Eclipse code editor. Improv (Chen and Guo 2019) produces slides for a live demo, linked to programmer’s editor where they can edit the code live.

These tools also highlight how code changes from one stage to the next. Many style the code to indicate what has changed since the previous stage (Ginosar et al. 2013; Mysore and Guo 2017; Mahoney 2018). One approach used in web-based tutorials today (Pombo 2019; Wattenberger 2019) is “scrollytelling,” where a program changes on one side of the screen as a reader scrolls through prose on the other side of the screen. The code pans, zooms, and highlights to focus the reader’s attention on how the code is changing at each stage.

Two tools generate mixed-media tutorials with segmented screencasts. Torta (Mysore and Guo 2017) segments screencasts following transitions from one GUI window to the next, and following command executions and text file edits. CodeMotion (Khandwala and Guo 2018) transforms existing screencasts into mixed-media tutorials, extracting frame-to-frame program differences from the screencast using computer vision, and then splitting these edits into “intervals” corresponding to changes made to a particular part of a file. Segmented screencasts are interspersed with the version of the code after each one, potentially making it easier for the reader to find segments of interest in the screencast.

Author experience. Studies have shown that these authoring tools both satisfy important needs, and may be further improved. Ginosar et al. (2013) observed in a study of their tool that seven authors could create multi-stage samples with the tool. Nearly every author needed to make edits to earlier stages, and did so successfully with the tool. Some authors would have appreciated alternative ways to edit code in multiple stages. The tool did not at the time provide the ability to make changes across stages by directly editing the code, and could not propagate edits forward in history, only backward. That said, participants found the tool useful, and appreciated the ability to switch between stages easily.

Do authors prefer creating textual tutorials or screencasts? It depends on the author and the design of the tool. In one study, authors using Torta (Mysore and Guo 2017) all preferred the mixed-media tutorials they created with Torta over the ones they created in Google Docs. They found that Torta allowed them to switch contexts less, encouraged them to explain the code more informally than they would in text, and automated much of the busy work of copying and pasting code and screenshots into the tutorial. In Oney et al.’s (2018) study of chat.codes, authors seemed to prefer recording screencasts instead of using chat.codes’ text-based interface if they felt comfortable speaking and programming at the same time. Otherwise, they preferred to produce tutorials with chat.codes.

Reader experience. Readers see benefit in the unique sorts of multi-stage samples produced by these tools. Students who had seen videos created using Storyteller (Mahoney 2018) reported that they valued the ability to see the evolution of a program rather than just the final, finished code. They also noted that the viewing interface, which showed both a screencast of the code being built and a simultaneous listing of comments in a side bar, could be somewhat overwhelming. In a study of tutorials generated by Torta (Mysore and Guo 2017), readers preferred those tutorials produced with Torta over those written by the same authors in Google Docs, noting that they were better-structured and more information-dense. In an elicitation study and a participatory design workshop with readers, Khandwala and Guo (2018) reveal even more interaction possibilities for presenting multi-stage

samples, like enabling inline code annotation within screencasts, and incorporating tables of contents and cross-video links.

Is it possible to multi-stage an existing program?

Perhaps sample programs could one day be transformed into multi-stage samples automatically, rather than requiring user labels or recordings of program editing sessions. In line with this vision, Sanchez et al. (2016) introduced algorithms that could split a code sample into stages, with each stage introducing new methods, and hide complex code blocks by folding the code contained therein. Along with these algorithms, Sanchez et al. provided an interactive browser for the multi-stage sample, and described a technique for inspecting code in the browser called “Multistaging to Understand.”⁵ When reading samples with the Multistaging to Understand inspection technique, 12 programmers from Upwork answered comprehension questions about code more accurately than when reading samples with a comparison inspection technique. Tools like these may one day be able to help authors split programs for which no programming history exists into multi-stage samples. In Chapter 6, we explore how a tool might be able to help a programmer take apart an existing program into a series of stages.

Limitations of multi-stage sample authoring tools

One challenge authors of tutorials face is keeping a variety of programming artifacts consistent with each other (Mysore and Guo 2017). In the process of creating a tutorial like the one from the beginning of Chapter 1, an author will create dozens of snippets and outputs, each of which represents a different stage in a multi-stage sample. The tools in the subsection above do not yet support authors in keeping the snippets and outputs that they derived from the multi-stage sample consistent with each other. In Chapter 6, a new type of notebook is designed for authoring multi-stage output-rich programming tutorials. This notebook propagates updates between all snippets and outputs in the tutorial.

Other tools that could support program distillation

When designing distillation tools, one can draw inspiration from a broad literature in human-computer interaction and software engineering. Algorithms and

⁵ Sanchez et al. (2016) define *distillation* as the “decomposition of a code example into chunks.” This is related to, but distinct from, the definition used in this dissertation.

interaction techniques from this literature could help authors select, simplify, supplement, and sequence their programs. This section reviews tools that help authors efficiently select code from source programs, clean messy programs, perform linked edits on programs and their outputs, and generate explanations.

Efficient code selection

For tasks ranging from debugging to copying and pasting code, programmers need more efficient mechanisms for selecting code from programs. Researchers have proposed both automated and interactive tools for extracting relevant lines of code from potentially large and complex programs.

Program slicing. One of the most widely-studied algorithms for efficient code selection is program slicing. Given a statement in a program, a slice is the subset of other statements required for that statement to run correctly. To slice a program, dependencies⁶ between statements are detected. A graph is built, where the nodes are statements, and the edges are dependencies. Then, a slice can be found for any statement by finding all other statements reachable from it in the graph, and removing the rest. Weiser initially introduced the idea of program slicing not as an algorithm, but as a description of what he thought programmers did when they debugged programs (Weiser 1979).

Many algorithms have since been devised to slice programs automatically (Silva 2012). One challenge to using slicers in practice, particularly for simplifying code, is that slices are often quite large, as computations become tangled with each other. It's not unusual for slices to be 10%, 20%, or sometimes even 50% of a program's total size (Binkley and Harman 2004). As such, several slicing techniques like dynamic slicing (Agrawal and Horgan 1990) and pruning (Zhang et al. 2006) have been devised to improve precision. Even if slicers are more precise with these modifications, they are hard to implement. Detecting dependencies between statements becomes difficult once you start considering, for instance, procedure calls (Horwitz et al. 1990) and concurrency (Jayaraman et al. 2005). As such, production-grade, user-facing program slicers are only available for a handful of programming languages, in a handful of code editors, such as Frama-C (Cuoq et al. 2012) and CodeSurfer (Anderson and Teitelbaum 2001).

⁶ In this dissertation, the word *dependency* is used to describe when one line of code depends on another, following the convention of other papers about interactive programming tools (e.g., Ko and Myers 2009; Holmes and Walker 2012; DeLine and Fisher 2015). The term *dependence*, without a *y*, is often used instead in the program analysis literature. The data structure containing all dependencies in a program is called by the name originally given to it by Ottenstein and Ottenstein (1984), the *program dependence graph*.

If the last 40 years of slicing history tell us anything, it is that we probably will not be seeing high-precision, production-grade, user-facing slicers for most languages anytime soon. It is more likely that developers will create slicers that are easy-to-implement, fast, and sometimes low-precision. This is the premise of Chapter 4, which assumes an imperfect slicer, and an author who can help resolve ambiguities when extracting a slice.

Other algorithmic selection techniques. Delta debugging (Zeller and Hildebrandt 2002) turns the problem of simplifying a program into a binary search problem. Given an input file that causes a program to crash, delta debugging will subdivide that file into lines, finding the subset of lines that cause the error by removing initially very large, and eventually very fine, subsets of lines. Another technique for selecting code of interest in a program is to analyze the program’s call graph. Given a program’s main function, all functions that are not transitively called by that function presumably aren’t necessary and can be removed (Tip et al. 1999).

Machine learning can be applied in the service of algorithmic code selection. For instance, recent work has shown that reinforcement learning can be used to train an agent that removes bloat from code by rewarding the agent when it produces minimal programs (Heo et al. 2018). If an agent is allowed to remove any arbitrary element from a program, it needs a comprehensive test set to run against the reduced program to check that it has not inadvertently introduced bugs.

Interactive aids for finding code. When a programmer is debugging a program, one question they may have is, *What are the lines of code that are responsible for this output I’m seeing?* Many tools have been designed to answer this question, in different programming environments, for different types of outputs. For instance, a tool can highlight lines of code as a program executes them (Brandt et al. 2010b; Burg et al. 2013; Oney and Myers 2009). It can filter the code for a program to only the lines that were run in a recent execution (Burg et al. 2015; Gross et al. 2010; Hibsichman and Zhang 2015; Hibsichman and Zhang 2016). It can also help a programmer trace through code backward from static outputs like console output and error logs (Ko and Myers 2009). The effects of these tools on a programmer’s efficiency can be pronounced. In one study of the Java Whyline, Ko and Myers (2009) found that programmers using the tools for debugging were successful in their task about three times as often and were about twice as fast.

Sometimes, programmers want to copy methods or classes that are tangled into a complex software system. Interactive tools can help programmers ensure they are copying a complete subset of the code. Gilligan (Holmes and Walker 2012), for instance, helps programmers review structural dependencies for code they plan to copy. The tool presents lists of dependencies to the programmer. The programmer

reviews the dependencies one by one, choosing which to copy, ignore, or replace with custom code. Chapter 4 introduces another tool that, like Gilligan, supports mixed-initiative, incremental code extraction.

Cleaning programs

Programmers clean their code to make it more readable, reliable, and maintainable. Researchers have proposed tools to help with these goals.

Refactoring. Refactoring is the process of applying behavior-preserving transformations to one’s code to improve its readability and maintainability. Routine refactorings include renaming variables and extracting code into functions (Fowler 2018). Tools for automated code refactoring are available in many popular programming IDEs (Murphy-Hill and Black 2008). Typically, a programmer interacts with refactoring tools via menus, wizards, and “quick fix” suggestions (see for instance those listed in the Eclipse user guide (Eclipse Foundation 2020)) anchored in the code next to entities that can be refactored.

Recently, researchers have explored how the user interface for refactoring tools can be redesigned to make refactoring functionality more discoverable, understandable, and flexible. Solutions have included the synthesis of refactoring rules from user demonstrations (Miltner et al. 2019), fine-grained code selection with in-situ refactoring menus (Hempel et al. 2018), enhanced visibility and control in refactoring previews (Barik et al. 2016), new visual languages for specifying program transformations (Boshernitsan et al. 2007), and refactoring by dragging and dropping program elements (Lee et al. 2013). Empirical studies of many of these tools (Barik et al. 2016; Boshernitsan et al. 2007; Hempel et al. 2018; Lee et al. 2013) show that they are usable and improve the experience of refactoring.

Simplifying programs. Even if they do not assist with the classical refactoring tasks described by Fowler (2018), a tool can help an author simplify a program in other ways. Amorphous slicing is a variant of program slicing that makes semantics-preserving transformations to slices as it extracts them (Harman et al. 2003). By this definition, the tool in Chapter 4 can be considered an interactive amorphous slicer. Researchers in program synthesis have explored how to synthesize simple programs by steering synthesis in the direction of programs that make use of common programming patterns (Fraser and Zeller 2011), and those that are estimated to be readable according to automated readability metrics (Daka et al. 2015). For decades, source code formatters have been used to improve the appearance of programs, and today new techniques are still being developed for related tasks like segmenting programs into meaningful blocks (Wang et al. 2014). Techniques have

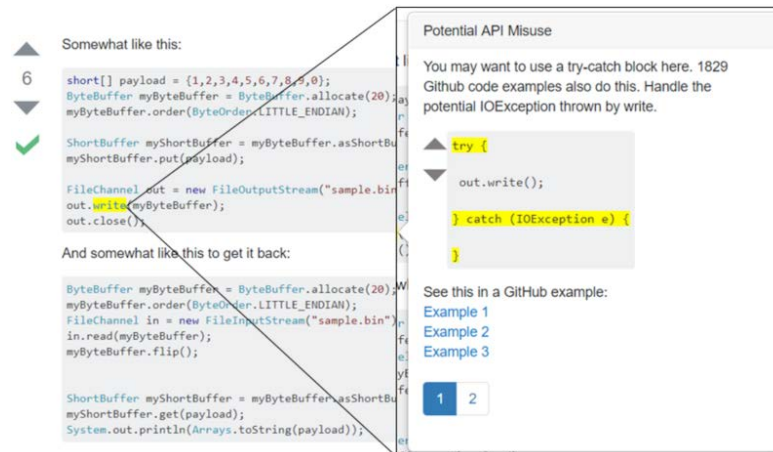


Figure 3.9: Fixing a code snippet with ExampleCheck. When a programmer is using a browser with the ExampleCheck extension, they are notified when the sample they are looking at might be unreliable. In this case, ExampleCheck proposes a try-catch block that an API call can be surrounded with to make the code more robust. Figure reused with permission (© 2018 IEEE, Zhang et al. 2018).

been devised for learning equivalent sequences of API methods (Goffi et al. 2014). All of these techniques may one day become components of interactive tools for cleaning code as part of the process of distillation.

Program repair. Oftentimes, a programmer’s code contains bugs and vulnerabilities of which the programmer is not aware. Quick fixes in IDEs often point out and recommend fixes to such errors. Tools for automated program repair detect bugs, vulnerabilities, and performance issues in software and propose fixes using heuristics, program synthesis, and machine learning (Le Goues et al. 2019).

Tools have been designed to assist with the repair of sample programs. The tools discover dependencies a sample needs to run (Horton and Parnin 2018), fix compilation errors (Terragni et al. 2016), suggest statistically-likely guards, exceptions handling, and resource management code (Glassman et al. 2018), and customizations and refactorings (Zhang et al. 2019). Interfaces to these tools help programmers fix sample programs at the moment when they find a sample on the web (Zhang et al. 2018; Zhang et al. 2019, see Figure 3.9) and when they search for samples (Glassman et al. 2018). Techniques for program repair could be incorporated into distillation tools to help authors create robust samples.

Linked edits to programs, documentation, and outputs

Linked editing enables programmers to make changes to one artifact that will propagate immediately to other artifacts. Three use cases of linked editing are

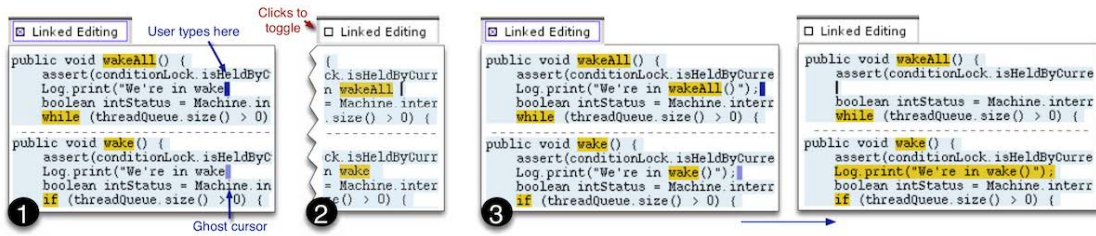


Figure 3.10: Linked edits of source code clones. With Linked Editing, programmers can establish links between copies of the same code. Differences between copies are highlighted in yellow. Any code highlighted in light blue will change in both copies at the same time (1). One copy can be temporarily unlinked from the other, such that changes from one will no longer propagate to the other (2). As a programmer changes an unlinked clone, the yellow highlights continuously update show how the clones differ (3). Figure reused with permission (© 2004 IEEE, Toomim et al. 2004).

discussed: keeping code clones consistent, keeping code and documentation consistent, and helping programmers evaluate their code as they write it.

Linked edits to source code clones. Programmers frequently copy their own code as they program (Lange and Moher 1989; Kim et al. 2004), creating what are known as *code clones*. One source of potential inefficiency and bugs is that when a programmer changes code in one place, they may forget to change the clones of that code elsewhere in the program. Toomim et al. (2004) introduced an editor-based interaction, *Linked Editing*, that supports the automatic propagation of edits between clones (Figure 3.10). Two blocks can be unlinked temporarily to allow an author to make changes to one, but not the other. Afterwards, the two can be linked once again to propagate edits between the parts of the clones that are still the same. Differences between the clones are highlighted.

Extensions to linked editing have been proposed to support different use cases. Juxtapose (Hartmann et al. 2008) employs linked editing to support rapid parallel prototyping of programs and exploration of parameter spaces. CloneBoard (Wit et al. 2009) provides specialized semantics for how to propagate changes between instances of clones on each edit action. CnP (Hou et al. 2009) establishes links between clones on a copy-and-paste interaction, supports refactoring within the bounds of a clone, and provides nuanced highlighting of differences between clones. Similarly, the tool introduced in Chapter 6 offers its own specialized linked editing semantics, linking snippets to the source programs they are taken from and allowing snippets' contents to diverge from previous snippets.

Linked edits to code and documentation. Programming documentation is notoriously out-of-date with the code it describes (Lethbridge et al. 2003; Uddin and Robillard 2015). Research tools have explored how to infer connections between code and

documentation so that the documentation may be kept up-to-date. These tools track references from the documentation to code elements like functions and classes (Subramanian et al. 2014). With such a link established, they can generate patches for the documentation as the code changes to remove outdated references (Dagenais and Robillard 2014) or change names of the API members referenced from the documentation (Lee et al. 2019).

Interactive tools can help programmers establish links in the other direction from code to documentation. Codetrail (Goldman and Miller 2009) automatically loads the documentation for selected code elements into a dedicated browser tab. It also detects when code in the editor is likely copied from an open web page, and creates bookmarks that will help the programmer later recall the provenance of that code. HyperSource (Hartmann et al. 2011) annotates lines in a programmer’s source code with histories of the pages that they were browsing as they edited those lines. CiteHistory (Fourney and Morris 2013) collects a log of the pages that a programmer browses as they compose an answer for a Stack Overflow question, making this history available to the programmer to be included as a list of references in their answer. These tools may one day be able to help authors make simultaneous changes to programs and their documentation, as well publish links to helpful resources alongside distilled programs.

Linked edits to code and outputs. The linked editing of code and outputs is a capability of many live programming environments. *Liveness* is the ability to modify a running program (Tanimoto 2013). Tanimoto introduced four levels of liveness in his paper about the VIVA system (Tanimoto 1990). With first level liveness, a user’s changes to a program have no effect on computation. At the second level, a user may submit their code to the system for execution. At the third level, a programmer’s edits trigger recomputation. Finally, at the fourth level, the system continually updates the display to show the time-varying results of processing input streams. In the literature, it is typically only systems that have third or fourth-level liveness that are called “live.” At both these levels, changes to code trigger changes to outputs. Liveness is a part of dozens if not hundreds of programming tools. This includes some notebooks (DeLine et al. 2015, Observable), recent research prototypes (Zhang and Guo 2017; Kang and Guo 2017), and prototypes of future general purpose code editors (McDirmid 2013).

A thought-provoking exemplar of live programming is Victor’s (2012) essay on *Learnable Programming*. Victor describes an environment designed to help programmers see and understand the execution of the program. It helps readers understand the vocabulary of the system by generating easy-to-read explanations of the program’s execution at the level of individual lines. Brushing and linking connects code expressions and the visual outputs they produce. A history of com-

putation is shown by superimposing snapshots of the visual output. The history can be explored through an interactive timeline with miniature representations of the program output at each step of execution. While these are just a few of the capabilities of Victor’s envisioned system, they hint at the rich expressive potential for tools that support linked editing of programs and their outputs.

Automated program explanation

A program that is difficult to read may be easy enough to explain, if described with the right words or pictures. This subsection provides a brief overview of techniques for automatically explaining and visualizing programs.

Generating natural language explanations. Natural language explanations can be generated for a wide range of software engineering artifacts, including blocks of code (Sridhara et al. 2011a), class diagrams (Burden and Heldal 2011), Java methods (Sridhara et al. 2010), unit test cases (Kamimura and Murphy 2013), method context (McBurney and McMillan 2014), parameters (Sridhara et al. 2011b), and classes (Moreno et al. 2013). Explanations can be generated through either rule-based methods (e.g., Sridhara et al. 2010) or machine learning models (e.g., Iyer et al. 2016). Some of these techniques have shown promising results. Sridhara et al., for instance, found that generated explanations of statements were judged to be accurate 92% of the time, adequate 85% of the time, and concise 96% of the time. Perhaps these techniques will one day be capable of generating explanations that can accompany distilled programs.

Software visualization. Visual tools have been used to aid in programming instruction for some time; Stasko et al. (1998) and Sorva (2012, pp. 140–185) present good reviews. As one recent example, Ou et al. (2015) produced visualizations of pointer-based data structures in the heap. PythonTutor (Guo 2013) is a programming visualization tool for CS education. The tutor is embeddable within a web page and supports simultaneous viewing of program source code, program execution, visual representations of Python objects, and program output. It has been incorporated into online textbooks about programming, including UC Berkeley’s introductory computer science textbook, *Composing Programs* (DeNero 2020).

One interesting class of software visualizations are those that can be shown *within the code*. A recent review called these visualizations *visual augmentations* (Sulír and Porubán 2017). Such visuals have been proposed to help programmers check their assumptions, recognize bugs, and understand their program’s execution (Lieber et al. 2014; Hoffswell et al. 2018). As one example, Theseus (Lieber et al. 2014) annotates the functions in a program with always-on counters that show

how many times each function has been called. Hoffswell et al. (2018) identified a design space of within-code visualizations that maps data properties like data type, temporality, and detail to suitable visual representations. They also recommend that such visuals are made comparable, salient, and unobtrusive. Augmentations like these are another instance of the ways that sample programs could be supplemented using future distillation tools.

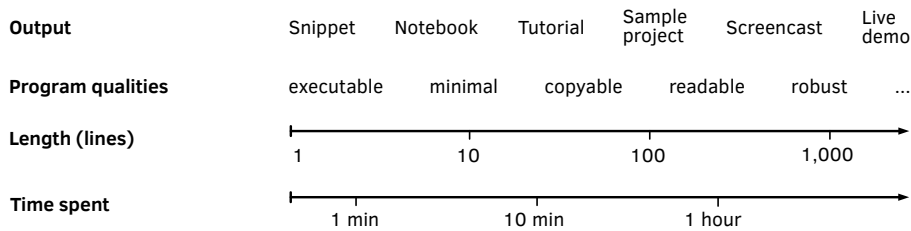
A design space for program distillation tools

How should tools help authors distill programs? A good answer to this question depends on what an author wishes to accomplish. To map out the space of capabilities that distillation tools might have, a design space is introduced. Design spaces, as described in the literature (Zwicky 1967; Jones 1992), help designers identify a wide range of design alternatives that might not come up during routine ideation.⁷ To build a design space, one determines a set of sub-problems a design must solve, and then identifies solutions to each of those sub-problems. Ideas for new systems come from combining the solutions to sub-problems in new ways.

Below, a design space for distillation tools is introduced. It begins with a survey of author goals. Then, solutions are described for each of the sub-problems of selecting, simplifying, supplementing, and sequencing code. Each sub-problem is considered one at a time, with a written description of design alternatives and a summary diagram. This design space is not meant to be exhaustive, but rather to consolidate many of the alternatives raised in the above discussion of related work. A diagram of the entire design space can be seen in Figure 3.11.

Goals. The usefulness of a distillation tool depends on how well it satisfies an author's goals. So, what is it the author wants to distill, and how do they wish to do so? Do authors want to produce a snippet, a tutorial, or some other type of program? Should the program be executable, readable, or robust? The author's *experience goals* matter as well (Cooper et al. 2007, p. 92), such as how quickly they wish to be finished, and whether they see distillation as an opportunity for creative expression, or instead as a practical task to be completed.

⁷ The creation of design spaces is common in human-computer interaction research. See for example Card et al. 1991, Fitzmaurice et al. 1995, and Hartmann et al. 2008.

**GOALS**

Tool basics. When a programmer decides to distill a program, would they like to do so within their usual code editor, a notebook, or some other dedicated environment? Each of these programming environments might already provide utilities for cleaning and formatting code. And what does interaction with the tool generally look like? Allen (1999) introduced four levels of mixed-initiative interaction among intelligent agents, which provide templates for the role that a distillation can take on when assisting an author:

- unsolicited reporting: notify the author of critical information as it arises.
- subdialogue initiation: start subdialogues to clarify or correct.
- fixed subtask initiative: take initiative to solve predefined tasks.
- negotiated mixed initiative: negotiate to determine initiative.

Is the authoring tool entirely autonomous, or does it engage in an on-going dialogue with the author according to one of these patterns?

Programming environment	code editor	notebook	dedicated editor	non-interactive	
Authoring role	Autonomous	Unsolicited reporting	Subdialogue initiation	Fixed subtask initiative	Negotiated mixed initiative

**TOOL
BASICS**

Selection. If the author needs to extract code from an existing program, how can the tool help them do so? An author might need to help the tool figure out what code is relevant, perhaps by selecting statements or outputs, or by interacting with the running program. The tool can then help the author extract code by slicing it, either creating a slice automatically or helping an author interactively expand the slice. The most sophisticated selection tools will help authors find code across many files written in many languages.

Author selections	Select statements	Select outputs	Interact with running program	None		
Slicing assistance	Slice	Interactive slice expansion	None			
Slicing domain	Interpreter history	One file, one language	Multiple files	Multiple languages	Library code	External services

SELECT

Simplification. If a programmer wants to make their program minimal and robust, how can the tool help? Authors might wish to remove complex statements, classes, or interfaces to external services. A tool could help authors simplify code by renaming identifiers, and replacing unwanted code with synthesized equivalent code, function and class stubs, or placeholders that a reader must replace. A distillation tool may not be able to decide on its own how best to simplify the code. In that case, the tool can ask the author to choose among options input with in-situ menus or design galleries (Marks et al. 1997). If there is an overabundance of ways the program might be simplified, the tool may need to rank the alternatives and recommend a small yet compelling set of them.

Help authors replace...	Statements	Classes	Entire programs	External services
Simplification techniques	Rename identifiers	Insert placeholders	Synthesize equivalent code	Generate stubs
# simplification options				

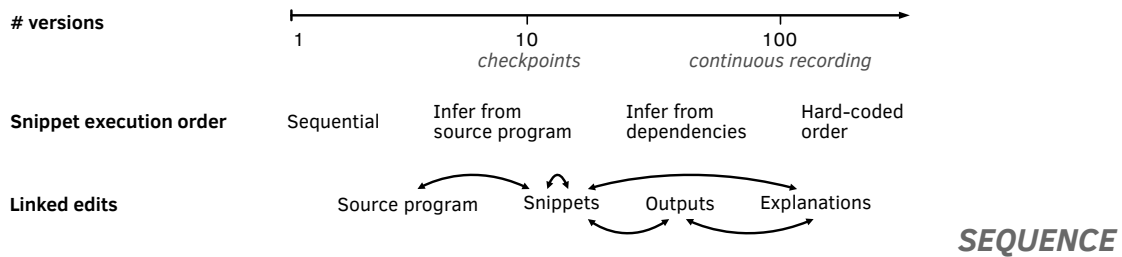
SIMPLIFY

Supplementation. If an author wants to make their program readable and robust, the program code might need to be embellished and the program may need to be explained better. A tool might help an author edit their program by identifying opportunities to insert guards and exception handling code where relevant. It could help authors explain programs by inserting realistic sample data and log statements that will expose the program’s state. Tools could also help authors generate assets like text explanations, diagrams, and screenshots of running programs. They could assist authors in creating interactive reading experiences like live program viewers where the code can be edited and executed, which would be time-consuming for the author to create without assistance.

Program embellishments	Guards	Program alternatives	Sample data	Exception handling	Log statements	
Assets	Rich text	Diagrams	Console output	Charts / graphs	Screenshots	Screen recordings
Interactivity for readers	Visualizations	Live editing	Exercises	Version browser		

SUPPLEMENT

Sequencing. If an author is creating a tutorial or a notebook, they will need to produce sequences of snippets, where each one potentially represents one stage in constructing a program. A tool can help authors maintain and simultaneously edit many versions of the same program at once. Editable histories can be created by asking authors to mark checkpoints in their program’s evolution, or by recording a detailed edit history as the author writes the program. A tool can help an author keep source programs, snippets, outputs, and explanations consistent by propagating edits between source programs and snippets, and by updating outputs and explanations as the code changes. How might a tool know how to regenerate outputs as snippets change? Snippets could be executed sequentially, in the order they appeared in the source program, in an order inferred from dependencies between snippets, or in some other order the author specifies.



This dissertation in the design space

This dissertation focuses on exploring a subset of the design space that has not yet been explored in prior work. With the above vocabulary for describing the capabilities of distillation tools, the unique designs of each of the tools in this dissertation can be described with greater clarity as follows. See Figure 3.11 for a visual representation of the capabilities of each tool within the design space.

Chapter 4. Snippet authoring via mixed-initiative selection and simplification. CodeScoop was designed to support efficient selection of code from existing programs and simplification of that code. Authors interact with the tool through mixed-initiative interaction for two reasons. First, because program slices may be imprecise, a mixed-initiative tool is suitable for helping an author declare what code they wish to keep in the program, and what code may be discarded. Second, sometimes there are multiple choices available to a programmer for how to simplify a program. For instance, they may remove a line of code, or replace it with any number of stubs. A mixed-initiative tool can expose these options to the author, while automating the tedious parts of distillation. CodeScoop in particular *initiates fixed subtasks* to automatically select code when there is only one suitable

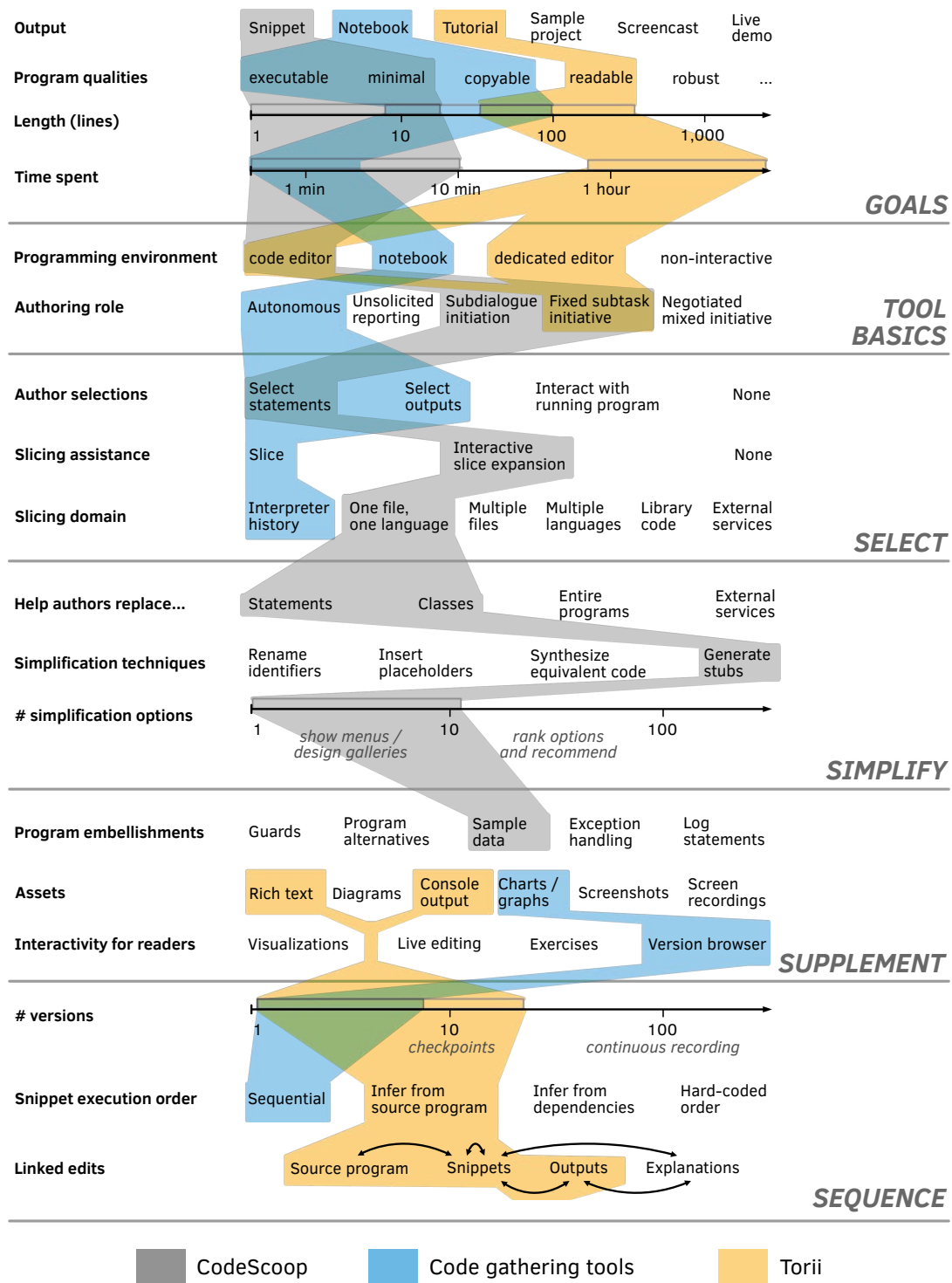


Figure 3.11: A design space of distillation tools, explored. Paths through the design space indicate the capabilities of each of the three tools introduced in this dissertation.

way to do so, and *initiates subdialogues* to ask authors for input when the author will know best whether code should be selected and how to simplify it.

Chapter 5. Notebook cleaning via direct selection of code and outputs. Code gathering tools were designed to support efficient selection of code from messy notebooks by helping authors extract ordered, reduced, complete subsets of cells that produce results of interest. The tools supplement the extracted code by packaging it into notebooks complete with selected results, and by showing the evolution of the cells and results in an interactive version browser. Code gathering tools are autonomous, not mixed-initiative. Their novelty lies in the direct and easy access they give authors to their programming history. Notably, the tools work by slicing not the code currently visible in the notebook, but the interpreter history.

Chapter 6. Tutorial authoring via flexible sequencing of snippets. Torii was designed to support the creation of flexibly-organized step-by-step programming tutorials. It combines the flexibility of organization present in WEB tools with the direct manipulation and liveness of computational notebooks. This required a new method to update outputs based on snippet content. Authors create snippets by selecting code in a source program. Each snippet, when added to the tutorial, implicitly creates a new version of the program with that snippet included. Outputs are generated by assembling the snippets above it in the order they appeared in the source program. This allows an author to provide syntactically-incomplete snippets, or to show snippets out of order, while keeping snippets and outputs consistent. Torii is mixed-initiative by necessity: the tool *initiates fixed subtasks* to regenerate outputs whenever the order or contents of snippets change.

Chapter 4. Snippet distillation

Mixed-initiative code selection and simplification

“Not that the story need be long, but it will take a long while to make it short.”

Henry David Thoreau

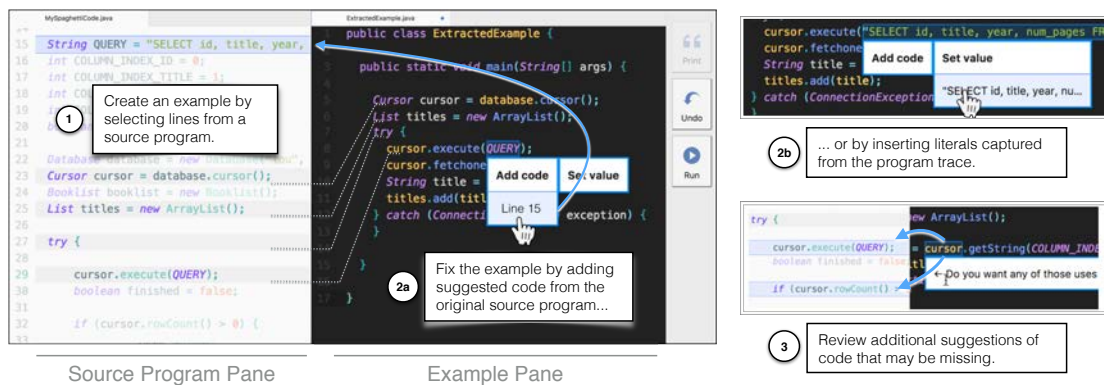


Figure 4.1: Extracting example code from existing code with CodeScoop. With CodeScoop, (1) a programmer selects a few lines they want to share from a source program, and CodeScoop helps them build them into a complete, compilable example. To help programmers make complete examples, CodeScoop detects 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.

Motivation

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 (Robillard and Deline 2011; Sacks 1994). Examples can demonstrate best practices for using particular APIs and confirm programmers' hypotheses about how things work (Robillard 2009). 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 (Ginosar et al. 2013; Kojouharov et al. 2004), searching for examples (Brandt et al. 2010a; Hoffmann et al. 2007; Stylos and Myers 2006), and integrating examples into one's own code (Oney and Brandt 2012; Wightman et al. 2012).

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 (Treude et al. 2011). Even for APIs that appear well-documented in online Q&A platforms, high coverage can take years to achieve and miss important topics (Parnin et al. 2012). Even if examples are available, they may not be self-explanatory: many lack important code required to run or understand them (Treude and Robillard 2017). 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 (Nasehi and Maurer 2010).

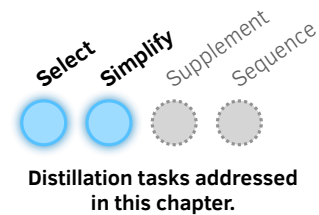
In this chapter, we aim to improve the state of the art in code example production. Good code examples are concise (Nasehi et al. 2012; Nykaza et al. 2002; Robillard and Deline 2011) and focused (Nasehi et al. 2012; Robillard and Deline 2011). Additionally, executable examples allow programmers to re-run 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 (Parnin et al. 2013). 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 (Tip 1995), 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 4.1). In this way, the tool supports the



mixed-initiative *selection* of code from a source program and *simplification* of the sample program. Once created, 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, this chapter contributes 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.

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.

Method

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 (referred to as F1–12 below). Participants were asked to create an understandable, executable code example from code they had recently written. The first five participants were asked to produce this code example as a blog post. We observed that the first five participants sometimes produced erroneous or untested code. With this observation, we realized that, without the proper tools, examples might be created that contain errors and do not run. We asked all remaining participants to only create a minimal, understandable, executable code example.

Participants authored code examples in a variety of languages, including PHP, Java, Python, JavaScript, Bash, C, and C#. Code examples demonstrated a variety of tasks, including querying a Salesforce database, and building a Chrome extension for circumventing news website pay walls. One participant (F12) unexpectedly brought code that had been written by someone else that they wished to simplify; because they followed a similar workflow as other participants, we include anecdotes from their experience as well.

Results

Authoring examples as iterative selection of code

When we cued the first five participants to produce a blog post that demonstrated a usage pattern with concise, easy-to-understand example code, all participants took an “additive” process to authoring example code, copying and pasting code from an existing project into a new file. Programmers copied and pasted individual lines (F2, F4, F6) and the contents of entire files (F1, F3) into a new buffer.

Participants who built a code example by moving code from an existing project faced two major obstacles. First, in some cases, it could be tedious to rebuild the original code’s project settings, dependencies, and environment for the code

example. For F6, this involved generating configuration files, setting paths to libraries, copying import statements, duplicating style files, and modifying the directory structure. All of these steps had to be completed before any example code could be tested. The same code could encounter unexpected bugs when run in a new location. For example, relative file paths resolved to the wrong location when the code was run in a different directory (F11).

Second, programmers introduced bugs in the process of copying, pasting, and modifying the code. Sometimes, important fragments of code were missing that caused the program to crash, like missing variable definitions or `return` statements (F2, F4). Programmers introduced bugs through manual edits that, while appearing innocuous, altered the program’s behavior in unexpected ways (F4, F10).

After the first participants faced obstacles when authoring example code additively, we asked F8–12 to author code examples “subtractively”, by removing code from a working project until they created a minimal code example. While participants no longer needed to configure the runtime environment for a new example, removing unrelated code and dependencies could still take tens of minutes (F10, F12). This confirmed for us that, for both approaches, the demonstrated authoring techniques were costly and tedious.

Modifying programs for reading and reuse

Participants omitted irrelevant implementation details to better highlight the focal code of an example. To reduce irrelevant detail, participants simplified strings to “Lorem Ipsum” (F5) and “hello world” strings (F7), and simplified the SQL queries to include only fields used by the final example code (F8). Return types of methods were changed from complex types to simpler types like lists (F8). Literal values were inserted to provide concise and reasonable default parameters (F8, F12). Participants commented out the names of argument parameters that would distract from those that the user should set, while leaving them present in the code to prevent compiler errors (F4), and annotated suggested values for arguments (F2, F4).

We observed that authors sometimes added code back in to make the example more usable. By adding code, authors could increase the visibility of execution, demonstrate best practices, and provide variations on examples. To make the code example’s internal computations more visible, participant F7 replaced a `return` statement with a `print` statement to make the code’s success obvious to someone who executed it, and F12 left in dozens of lines dedicated to printing out updates to a machine learning model during training. F11 added explicit `if` statements to check whether method return values were null pointers and empty lists, to make

Authors made examples by...	Tools should help authors...
Copying the original code and pasting into example editor	<ul style="list-style-type: none"> • Create examples from text selections • Add lines from original code at any time
Replacing variables with meaningful literal values	<ul style="list-style-type: none"> • Review and insert literal values that preserve program behavior
Tweaking comments and code format for readability	<ul style="list-style-type: none"> • Directly edit code to add comments, group lines, and add <code>print</code> statements
Making examples could be time-consuming because...	Better tools could...
Authors left out code	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Constrain manual code edits • Enable early and frequent testing
It took time to remove irrelevant code	<ul style="list-style-type: none"> • Start from a blank file • Omit code except for explicit code selections and necessary fixes

Figure 4.2: 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.

clear to readers that the API calls might return empty values. F4 produced three versions of one code example, each of which utilized different test data and showed progressively more advanced usage of an audio API.

These observations and our review of the literature on code examples led to design recommendations for improving the user experience of extracting code examples from existing code (Figure 4.2).

Design motivations

We designed CodeScoop as a tool to help authors extract examples from existing code. 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.

A demo of CodeScoop

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 artifact¹ accompanying this chapter to see the full “scooping” process.

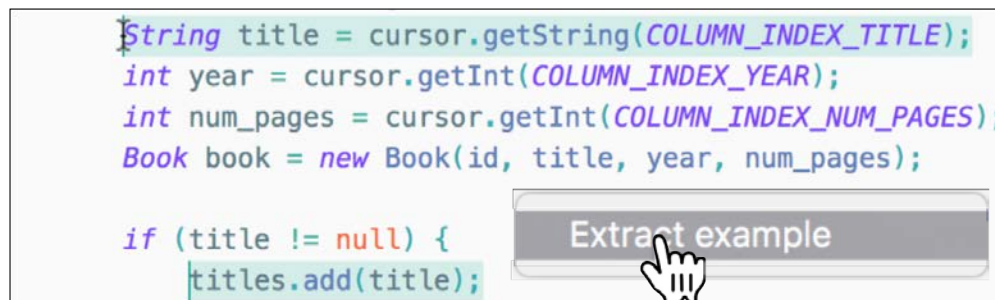
¹ See the project web page at <https://codescoop.berkeley.edu>

Prologue: An unexpectedly useful programming pattern

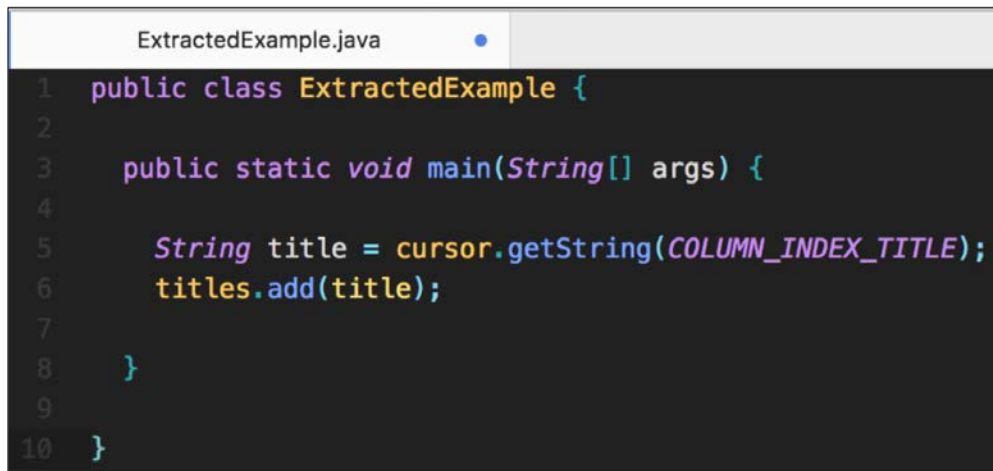
Lou is a programmer working on a web application. She needs to write code to query records from a database using an archaic, poorly-documented API. After reading source code, inspecting the runtime state of the database objects, and overcoming numerous compiler errors, she develops the code that she 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 steps: 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.



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 4.1). 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:



```
ExtractedExample.java
1 public class ExtractedExample {
2
3     public static void main(String[] args) {
4
5         String title = cursor.getString(COLUMN_INDEX_TITLE);
6         titles.add(title);
7
8     }
9
10 }
```

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.



```
6 titles.add(title);
```

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 4.1.2a and Figure 4.1.2c).

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.

The screenshot shows a code editor with the following code:

```

20     boolean DEBUG = true;
21
22     Database database = new Database("lou",
23     Cursor cursor = database.cursor();
24     Booklist booklist = new Booklist();
25     List titles = new ArrayList();
26

```

Line 25 is highlighted. A context menu is open over it, showing options: "Add code", "Stub out", and "Line 5". A dashed yellow arrow points from the menu to the code.

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.

The updated code in the editor is:

```

7     List titles = new ArrayList();
8     String title = cursor.getString(COLUMN_INDEX_TITLE);
9     titles.add(title);

```

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.

The screenshot shows the code editor with the following code:

```

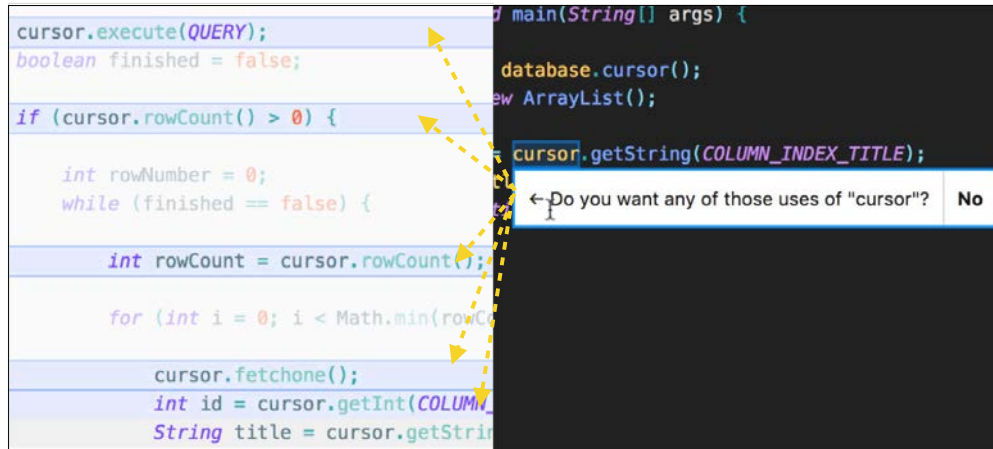
String title = cursor.getString(COLUMN_INDEX_TITLE);
String title = cursor.getString(1);
titles.add(title);
} catch (ConnectionException excep
}

```

The value `1` in the second line is highlighted. A context menu is open over it, showing options: "Add code", "Set value", and a hand icon.

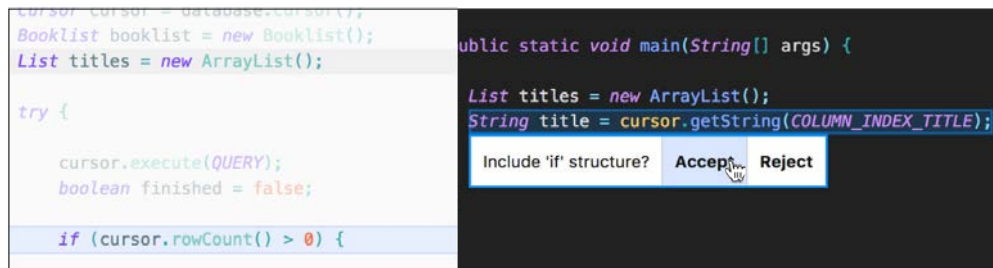
Checking for omissions by reviewing previous variable uses. Even if all of the variables in a program are defined, this doesn't mean that the program is correct—it might be missing important modifications on already-initialized objects. To make sure Lou isn't leaving out anything important, CodeScoop 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.

Defining 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.

The screenshot shows a code editor with two panels. The left panel displays Java code for a `Database` class. A yellow dashed arrow points from the `new Database().cursor();` line in the right panel to the `private class Database {` line in the left panel. The right panel shows a context menu with options: **Add code**, **Stub out**, and **Preview Stub 1**. The `Preview Stub 1 option is highlighted with a mouse cursor.`

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 4.1). 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.

Implementation

Code extraction with the “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 4.3). 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 4.4). These modules are described below.

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:

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 SimpleDefUseAnalysis program from the Soot (Vallée-Rai et al. 1999) system. 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.

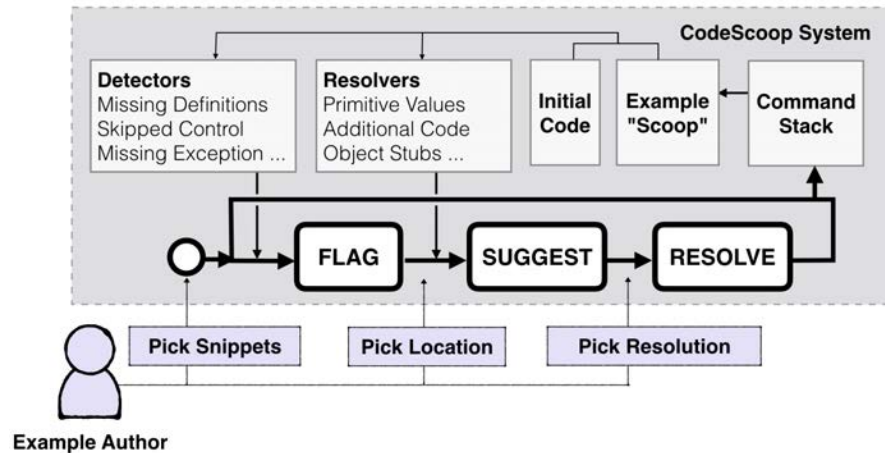


Figure 4.3: Iterative correction of incorrect example code. The CodeScoop system *flags* opportunities to complete and expand the code, *suggests* lists of valid resolutions for completing the code, and *resolves* completions by modifying an abstract example model called the *scoop*.

CodeScoop also runs an ANTLR-generated (Parr and Quong 1995) 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.

Missing previous uses of a variable. When a user adds a variable use and its definition, CodeScoop collects all past uses of the variable that occur between the use and definition, and which are not yet in scoop. These previous uses are presented to the user all at once.

Missing exceptions for a function call. Using Soot as a parser, we locate all function calls in the source program. For each call, we use the Java Reflections API² to find all exceptions the method can throw. An ANTLR parser is run on the source program to find `try-catch` blocks and the exceptions they catch. When a line is added to the scoop with an exception-prone function, CodeScoop checks to see whether (a) the scoop already throws the exception or a superclass of the exception or (b) the scoop includes a `try-catch` block surrounding the function

² <https://docs.oracle.com/javase/tutorial/reflect/index.html>

CodeScoop analyzes the **source program** and its **execution trace** to detect when example code is incomplete, and to suggest fixes.

The user interacts with CodeScoop to complete the example. The user **picks fixes for errors** and **accepts or rejects optional inclusions**.

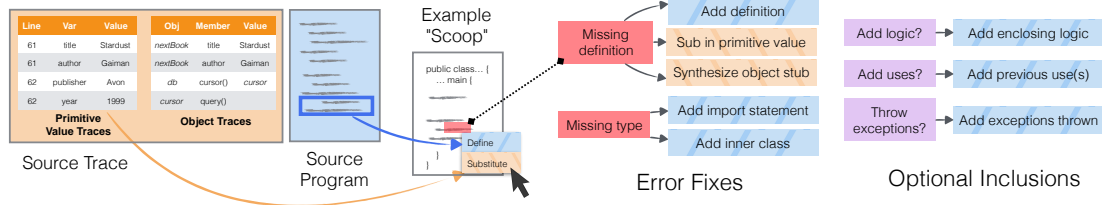


Figure 4.4: Suggesting fixes and code that complete a “scoop.” Given a set of text selections, CodeScoop detects what’s missing from an example. Whenever it detects an error with multiple resolutions, it prompts the user with a dialogue. A collection of static and dynamic analysis modules enable CodeScoop to detect missing definitions and types, propose fixes to errors, and to suggest optional code that might belong in a scoop.

call that catches that exception. If neither of these is true, CodeScoop reports that this exception must be caught or thrown.

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 4.4). 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](#). 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 imports 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.

Throwing exceptions for exception-prone function calls. CodeScoop searches for an `import` statement that defines either the exception or one of its superclasses. When a relevant type is found in the import statements, CodeScoop recommends that this exception type is thrown from the `main` method. If a user accepts the suggestion, the exception type is added to the `throws` clause of the `main` method.

Suggesting stubs for undefined object variables. CodeScoop extends the idea of replacing variable names with literal values to complex data types, by generating “stubs” with identical runtime behavior to object variables. While this feature is currently experimental, we propose it as one possible solution. To create stubs, an additional debugger virtual machine is launched. CodeScoop tracks every property access and every method call on every object defined in the source program, and logs all values these properties and methods returned. These variables and method calls can return other objects, which get tracked as well. CodeScoop generates stubs as classes that replicate the recorded behavior given the same order of property accesses and method calls.

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.

Generating an example program from the “scoop” data structure

Generating the example text from a list of selections and user choices involves instantiating a hierarchical string template. First, the text for all text selections, stored as numerical “ranges” that indicate the character offset of the text selections in the source program, is retrieved from the source program. These texts are joined in a temporary buffer, in the order that they appeared in the original code. The position of each of these selections is “marked” in the example text editor. These markers will keep track of the selection’s offset in the example text as it is built up, and will be used later for highlighting errors and inserting literal values.

Next, the snippets are wrapped in the `main` function of a class called `Extracted-Example`. Object stubs are generated from inner classes and added above the `main`

function. All inner classes and local methods that the user included are added as `static` members of the class below the `main` function. Finally, the generated example code is automatically indented.

Once all selections and stubs have been added, literal values are inserted into the example, in the positions tracked by the “markers” of individual text selections. If the program is in the *flag* state, it then highlights errors for a user to choose from. If in the *suggest* state, it highlights the chosen error and displays a menu for previewing and accepting resolutions.

Implementation specifics and 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 and 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 implemented (e.g., using [Chen et al.’s \(2014\)](#) bytecode analysis approach) 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.

In-lab usability study

Method

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:

- *RQ1. 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?

- **RQ2. 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?
- **RQ3. 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.
- **RQ4. 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 (Tip 1995). Do “scoops” offer advantages over such slices? Or do they give work to humans that could be done by an algorithm?

To answer these questions, the study comprised four stages:

Example extraction tasks. 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 used by participants in the formative study, 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.

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).

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 4.5). Difficulty varied based on the type of suggestion. Deciding whether to throw an exception was not difficult—if the exception wasn't handled,

How difficult was it for you to decide whether to accept these suggestions when you were making an example?

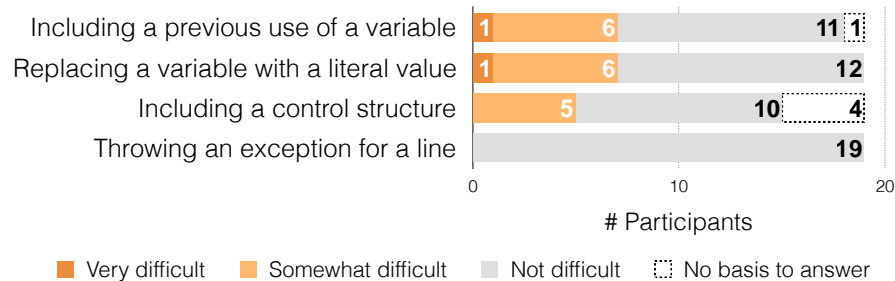


Figure 4.5: Not all choices in code extraction are easy. While some of CodeScoop’s suggestions weren’t difficult to consider (throwing exceptions for exception-prone lines of code), others required some thought (e.g., considering whether to include a previous use of a variable).

the code just wouldn’t compile (P13). It was trickier to make decisions about whether to include other uses 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:

“[With the text editor,] 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 ($\sigma = 1.96$), and 9.5 minutes with the baseline text editor ($\sigma = 1.52$), 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 ($\sigma = 2.64$) 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

```

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) {
}

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);

```

(a) Scoop 1: Contains try-catch block, and checks rowCount.

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

Figure 4.6: 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.

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 4.6 for one case).

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 4.7). 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 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 twelve³ 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 [in the slice] 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).

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

³ 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.

Variable	Add Code	Insert Literal	Variable	Add Code	Insert Literal
COLUMN_INDEX_ID	18	5 6 15	arg0	3 4 8 19	16
COLUMN_INDEX_NUM_PAGES	18	5 12 15	priceInt	3 4	
COLUMN_INDEX_TITLE	18	5 6 12 15	query	3 4 8 16 19	7 9
COLUMN_INDEX_YEAR	18	5 12 15	<i>Task 2</i>		
num_pages	5 18		arg1	2 10	14
QUERY	5 17	6 18	destination	1 2 10 13 14	11
<i>Task 1</i>			messageHtml	1 2 10 14	11 13
			password	1 2 13	11 14
			sslFactoryClass	1 2 10 11 13	14
			username	1 2 13	11 14
			<i>Task 3</i>		

Figure 4.7: 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.

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 participant 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.

Conclusions

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.

Limitations and extensions

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](#)); and “bubbles”-based code navigation paradigms ([Bragdon et al. 2010](#)).

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.

Chapter 5. Notebook distillation

Cleaning messy computational notebooks

“But the beginning of things, of a world especially, is necessarily vague, tangled, chaotic, and exceedingly disturbing. How few of us ever emerge from such beginning! How many souls perish in its tumult!”

Kate Chopin

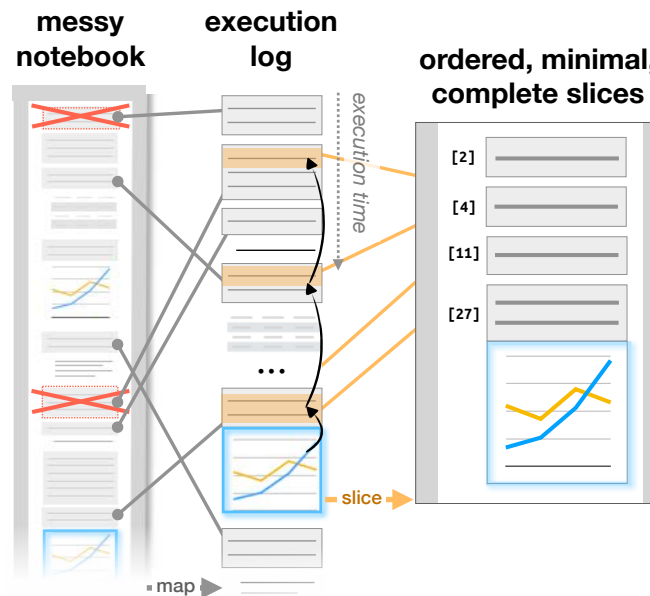


Figure 5.1: Code gathering tools help analysts manage programming messes in computational notebooks. The tools map selected results (e.g., outputs, charts, tables) in a notebook to the ordered, minimal subsets or “slices” of code that produced them. With these slices, the tools help analysts clean their notebooks, browse versions of results, and discover provenance of results.

Motivation

Data analysts often engage in “exploratory programming” as they write and refine code to understand unfamiliar data, test hypotheses, and build models (Kery and Myers 2017). For this activity, they frequently use computational notebooks, which supplement the rapid iteration of an interpreted programming language with the ability to edit code in place, and see computational results interleaved with the code. A notebook’s flexibility is also a downside, leading to messy code: in recent studies, analysts have called their code “ad hoc,” “experimental,” and “throw-away” (Kandel et al. 2012), and described their notebooks as “messy” (Kery et al. 2018; Rule et al. 2018c), containing “ugly code” and “dirty tricks” in need of “cleaning” and “polishing” (Rule et al. 2018c).

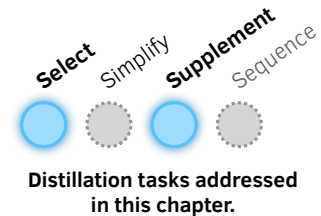
In essence, a notebook’s user interface is a collection of code editors, called “cells.” At any time, the user can submit code from any cell to a hidden interpreter session. This design leads to three types of messes common to notebooks: disorder, where the interpreter runs code in a different order than it is presented in the cells; deletion, where the user deletes or overwrites the contents of a cell, but the interpreter retains the effect of the cell’s code; and dispersal, where the code that generates a result is spread across many distant cells. For the millions of users of notebooks (Kelley and Granger 2017), such messes are quite common: for instance, nearly half of public notebooks on GitHub include cells that were executed in a different order than they are listed (Rule 2018). Messes make it difficult for an analyst to navigate and understand their code, and to recall how results (e.g., charts, tables) were produced. Messes also make analysts reluctant to share their analyses with stakeholders and collaborators (Kery et al. 2018; Rule et al. 2018c).

In this chapter, we aim to improve the state of the art in tools for managing messes in notebooks. We introduce a suite of interactive tools, *code gathering tools*, as an extension to computational notebooks. The tools afford analysts the ability to find, clean, and compare versions of code in messy notebooks. They build on a static program analysis technique called program slicing (Weiser 1981), which answers queries about the dependencies among a program’s variables. With code gathering tools, an analyst first selects a set of analysis results, which can be any cell output (e.g., charts, tables, console output) or variable definition (e.g., data tables, models). Then the tool searches the execution log—an ordered history of all cells executed—to find an ordered, minimal subset or “slice” of code needed to compute the selected results (Figure 5.1).

This chapter makes two contributions. The first contribution is the design and implementation of code gathering tools. Specifically, the tools highlight dependencies used to compute results, to help analysts find code they wish to understand,

reuse, and rewrite in cluttered notebooks. They provide ordered, minimal code slices that can serve as succinct summaries of analysis activity or starting points for branching analyses. Additionally, they archive past versions of results and allow analysts to explore these versions, and the code slices that produced them.

The tools therefore support two of the distillation tasks identified in Chapter 2: *selecting* code from messy notebooks, and *supplementing* the extracted code with selected results and histories of how the results evolved over time. Code gathering tools are implemented as an extension to *Jupyter*, a popular notebook with millions of users (Kelley and Granger 2017). The extension is available for use as a design artifact and as a practical tool for exploratory data analysis in notebooks.



The most important idea behind the interaction design of code gathering tools is *post-hoc mess management*—that tools should allow analysts to easily find, clean, and compare versions of code in notebooks, regardless of whether they have followed a disciplined strategy to organize and version their code. Past tools for cleaning code often require effort: annotating cells with dependency information (Jupyter Contrib Team 2020), folding and unfolding cells (Rule et al. 2018a), and marking and tagging lightweight versions of snippets (Kery et al. 2017). With code gathering tools, history is stored silently, and tailored slices of code are recalled on-demand with two or fewer clicks.

Our second contribution is a qualitative usability study providing insight into the uses and usability of code gathering tools for managing messes in notebooks. 12 professional data analysts used the tools in an in-lab study to clean notebooks and perform exploratory data analysis. We found that affordances for gathering code to a notebook were both valued and versatile, enabling analysts to clean notebooks for multiple audiences, generate personal reference material, and perform lightweight branching. We also refined our understanding of the meaning of “cleaning,” and how code gathering tools support an important yet still incomplete set of tasks analysts consider to be part of code cleaning. This study confirmed that analysts desire tools that help them manage exploratory messes, and that code gathering tools provide a useful means to manage these messes.¹

¹ See the project web page, <https://microsoft.github.io/gather/>, for installation instructions for the extension, and study materials.

Design motivations

We conducted formative interviews with eight data analysts and builders of tools for data analysis at a large, data-driven software company. During the interviews, we proposed several extensions to the notebook interaction model. Analysts expressed the most enthusiasm for tools to help them clean their results, and explore past variants of their code. These conversations and a review of the related literature yielded several key ideas that guided our design of notebook cleaning tools. We refer to the analysts as D1–8 below.

Post-hoc management of messes. Analysts have diverse personal preferences of whether and how to organize and manage versions of code. The analysts we spoke to each had their own workarounds, like keeping cells ordered so they always reproduce the visible results (D7, D8), copying useful snippets to external files (D4), and assigning dataset variables new names every time they transform them to avoid overwriting the original data (D6). Some code organization strategies conflict with others: some analysts clean their notebooks as they write it, while others preserve a record of everything they have tried (Kery et al. 2018)—though you cannot do both in current notebooks. One analyst noted that you don’t always know if you are creating versions of code until you already have (D7). We decided code gathering tools should assist analysts regardless of whether they think to organize their code, and whether they prefer to overwrite or save copies of old code. The tools silently collect history, and provide access to the code that produced any visible result.

Portability of gathered code. Analysts reuse a notebook’s code in that notebook, other notebooks, and scripts (Kery et al. 2018). The analysts we spoke to wanted tools to help them reuse code in new notebooks (D7), to apply old notebooks’ analyses to new data (D8), and to export code to other files (D4, D5). We designed our tools to make it equally easy to gather code to new notebooks, cells, and lines of text.

Query code via direct selection of analysis results. Prior research shows that programmers frequently look to program output when searching for code to reuse (Ragavan et al. 2016). In notebooks, visual results break up walls of monospace text, providing beacons. We anticipated that selections of results would provide the most direct method for accessing relevant history.

A demo of code gathering tools

To convey the experience of using the code gathering tools in Jupyter Notebook, we describe a short scenario. Consider an analyst, Dana, who is performing exploratory data analysis to understand variation and determiners of quality of a popular consumer good—chocolate. This section shows how code gathering tools could help her find, clean, and compare versions of code during data analysis.

Prologue: A proliferation of cells

Dana starts her analysis by loading a dataset, importing dependencies, and filtering and transforming the data. She writes code to display tables so she can preview the data. To better understand key features of the data, she builds a model to predict chocolate quality from the other features. Through experimentation, she tailors the model parameters to learn more about the features. Throughout the analysis, she makes messes, overwriting old code, deleting code that appears irrelevant, running cells out-of-order, and accumulating dozens of cells full of code and results. Dana starts to have trouble finding what she needs in the notebook.

Finding the code that produces a result

After several hours building and testing models, Dana is satisfied with a version of the model, but then realizes there may be a problem with the model. One of the numeric fields contains erroneous values. Although Dana wrote code to fix these values, she cannot remember if she ran this code on the dataset that was used to trained the model.

Because she has installed code gathering tools, Dana sees all variable definitions (data frames, models, etc.) highlighted in blue and all visual outputs (console output, tables, figures, etc.) outlined in blue. She clicks on the results of classification (a variable named `predictions`) and then all lines that were used to compute the variable's value are highlighted in light purple (Figure 5.2). Dana scrolls through the sprawling notebook to browse the highlighted lines, skipping over long sections of irrelevant code and results. She finds the code that transforms the percentage data, namely, a cell defining the function `normalizeIt` and the cell after it. Because these lines are highlighted, Dana knows that she cleaned the column of unclean values before classification.

The screenshot shows a Jupyter notebook with several code cells. The cells are:

- In [206]: `def normalizeIt(percent):` followed by an if-statement and `return percent`.
- In [207]: `df['CocoaPercent'] = df['CocoaPercent'].apply(normalizeIt)`
- In [209]: `df['Rating'] = (df['Rating'] * 100).astype(int)` followed by `df['Rating'].head(5)`.
- In [216]: `X = df.drop('Rating', axis = 1) #Features`, `y = df['Rating'] #Target Variables`, and `X_train, X_test, y_train, y_test = train_test_split(X, y, test_si`.
- In [220]: `dtree = DecisionTreeClassifier(max_depth=12)` followed by `dtree.fit(X_train, y_train)`.
- In [221]: `predictions = dtree.predict(X_test)`

 The code in cells 206, 207, 209, 216, and 220 is highlighted in light purple. Black arrows point from the `normalizeIt` function in cell 206 to its use in cell 207, from the `df['Rating']` assignment in cell 209 to its use in cell 216, and from the `X_train, X_test, y_train, y_test` assignment in cell 216 to its use in cell 220. A hand cursor is shown over the `predictions` variable in cell 221.

Figure 5.2: Finding relevant code with code gathering tools. With code gathering tools, an analyst can click on any result, and the notebook highlights in light purple just those lines that were used to compute the result or variable. The highlights appear throughout the notebook (which is condensed in the figure). Black arrows have been added in the figure to indicate the data dependencies that cause each line to be included in the highlighted set.

Removing old and distracting analysis code

Dana now has a notebook with a model that she likes—and much more code she no longer needs (Figure 5.3.1). Now that Dana knows what her data looks like and has a working set of data filtering, data transformation, and model training code, the code to visualize the data and debug the APIs will just get in the way. Dana decides to clean her notebook to a state where it only has the useful model-building code.

To clean the notebook, Dana clicks on a few results she wants to still be computed in the cleaned notebook, namely, the classification results in the `predictions` variable and a histogram showing the range of chocolate qualities used to build the classifier (Figure 5.3.2). Dana gets a sense of the size of the final cleaned notebook by looking at which lines in the notebook are highlighted as she selects each result. Then, Dana clicks the “Gather to Notebook” button (Figure 5.3.3), which opens a new notebook with the definition of `predictions`, the bar chart of chocolate quality, and the other code needed to produce these two results. The new, cleaned notebook has 16 cells, instead of the 47 in her original notebook. It contains the bar chart and omits 28 other visual results in the original. This reduces the overall size of the notebook from 13,044 to 1,248 vertical pixels in her browser, which is

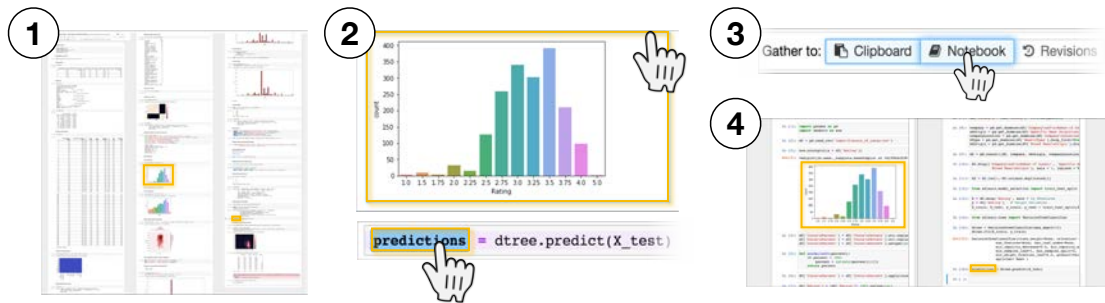


Figure 5.3: Cleaning a notebook with code gathering tools. Over the course of a long analysis, a notebook will become cluttered and inconsistent (1). With code gathering tools, an analyst can select results (e.g., charts, tables, variable definitions, and any other code output) (2) and click “Gather to Notebook” (3) to obtain a minimal, complete, ordered slice that replicates the selected results (4).

much easier to scroll through when editing the code (Figure 5.3.4). This cleaned notebook is guaranteed to replicate the results, as the tool reorders cells and resurrects deleted cells as necessary to produce the selected results. Dana verifies that running this notebook start-to-finish indeed replicates the chosen predictions and bar chart.

Reviewing versions of a result and the ordered, minimal code slices that produced them

To build a better predictor, Dana has been experimenting with different parameters to a decision tree classifier, like its maximum allowable depth and the minimum samples per branch. Dana remembers that she had previously created a simple, shallow decision tree with promising performance, but has not yet found a model with better performance.

With code gathering tools, Dana can summon all past versions of her classifier’s results and compare the code she used to produce these results. To do this, she clicks on a result—namely, a confusion matrix which visualizes the accuracy of the decision tree for each class—and then on the “Gather to Revisions” button. This brings up a version browser (Figure 5.4). Here, Dana sees all the versions of the result, arranged from left to right, starting with the current version and ending with the oldest version. Each version includes the relative time the result was computed, the code slice that produced that version and the result itself.

Scrolling horizontally to access older versions, Dana finds several examples of decision trees with comparatively good accuracy. Differences from the current version of the code are shown with bold text and a colored background. Dana finds the model she is looking for—a shallow tree with good performance. The

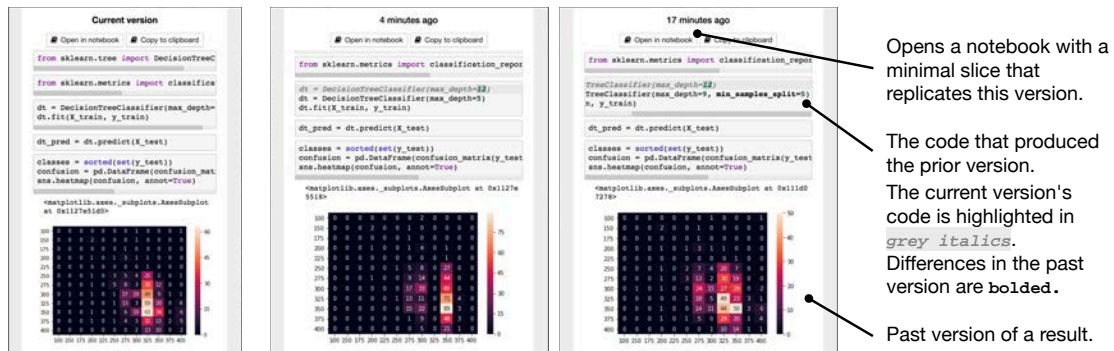


Figure 5.4: Comparing versions of a result with code gathering tools. When an analyst executes a cell multiple times, code gathering tools archive each version of the cell. When the analyst chooses the cell's output—say, the confusion matrix shown above—and clicks “Gather to Revisions,” a version browser appears that lets them see all versions of that output, compare the code slices that produced each version, and load any of these slices into a new notebook, where the version's results can be replicated.

code that produced this version can be copied as cells or text to the clipboard, or opened as a new notebook that replicates that version; Dana opens a notebook with this version so she can refer back to it later.

Cleaning finished analysis code

Dana finished her data analysis and wants to share the results with an analyst on her team who can check her results and suggest improvements. However, the notebook is once again cluttered with code that would distract her colleague. While Dana wants to save her long and verbose notebook for her personal use later, she also wants a clean and succinct version of the notebook for her colleague. She chooses the prediction results of her model, clicks “Gather to Notebook,” and saves the generated notebook to a folder shared with her colleague.

Exporting analysis code to a standalone script

After refining her analysis with her colleague, Dana wants to export a script that can be packaged with an article she is writing, so that others can replicate her results in their preferred Python environments. To do this, Dana selects the code that produces the results she wants her script to replicate, clicks “Gather to Clipboard,” and then pastes the gathered code into a blank text file. This script replicates the results Dana produced in her notebook.

Implementation

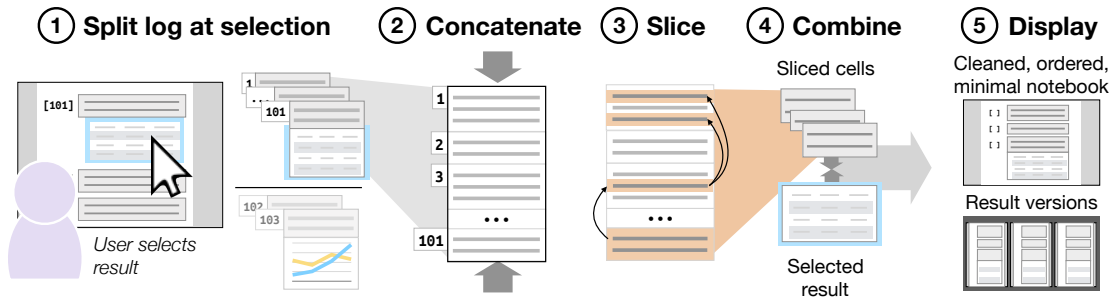


Figure 5.5: Implementation of code gathering. When an analyst wants to gather the code that produced a result, the code gathering backend splits the log of executed cells at the last cell where the analyst clicked a result, and discards the other cells (1), concatenates the text from the remaining cells into a program (2), slices the program using the analyst’s selections as a slicing criterion (3), combines the sliced cells with the selected results if they are code outputs (4), and displays these cells in a notebook or a version browser (5).

A computational notebook uses an underlying language interpreter. At any time, an analyst can submit any cell’s code to the interpreter, in any order. The results that the interpreter produces and that the notebook displays depend on the order in which the analyst submits cell code. Hence, in the notebook context, a notebook’s “program” is not the content of the notebook’s cells, but the content of the cells that the analyst runs, in the order in which the analyst runs them. We call this the *execution log*.

We define code gathering as the application of program slicing to an execution log to collect ordered, minimal subsets of code that produced a given result. Program slicing is a static analysis technique wherein, given a target statement (called the slicing criterion), program slicing computes the subset of program statements (called the slice) that affect the value of the variables at the target statement (Weiser 1981). In the notebook context, the variables/outputs that an analyst selects are the slicing criteria, and the gathered code is the slice. We implemented code gathering as a Jupyter Notebook extension with roughly 5,000 lines of TypeScript code. Our implementation supports notebooks written in Python 3. The details in this section could serve as a conceptual template for tool builders seeking to support code gathering for notebooks in other Python-like languages like Julia and R.

Collecting and slicing an execution log

To find the code that produces a result, the tools first need a complete and ordered record of the code executed in the notebook. We build such a record, the “execution log,” by saving a summary of each cell as it is executed. A cell summary contains two parts: first, the cell’s code, which will be joined with the code of other cells into a temporary program used to find code dependencies; second, the cell’s results, which can be used as slicing criteria, and shown in a version browser as the output of running that cell.

The code for some cells, if included in the execution log, will cause errors during program slicing. Namely, if the code contains syntax errors, the temporary program used during dependency analysis will fail to parse; if it raises runtime errors, a slice containing that cell might raise the same error. Therefore, cells with syntax errors and runtime errors are omitted from the log. Ignoring cells with *parse errors* is consistent with Jupyter’s semantics: if an executed cell contains any parse errors, all of its code is ignored by the interpreter. Ignoring cells with *run-time errors* is inconsistent with Jupyter’s semantics, in that the interpreter will run the statements up to the point where the error occurs. This limitation does not cause problems in practice, since analysts typically correct such errors and re-run the cells.

Next, we slice the execution log to produce code slices that replicate results. When an analyst selects results in a notebook, they specify slicing criteria. When they select a variable definition, they add the statement containing the variable definition as a slicing criterion. When they select a cell’s output, they add all statements from that cell.

To slice the execution log, there must first be a “program” to slice. We build such a program by filtering the log to exclude the cells that were executed after the cells containing slicing criteria: these cells won’t be included in the slice, and would unnecessarily slow down the slicing algorithm. Then, the program is built by joining the text of the remaining cells, in the order they were executed (Figure 5.5.1–2). This program may include the code of a single cell more than once, e.g., if the cell was executed twice to compute the chosen result.

Finally, we slice the program (Figure 5.5.3). We implemented a standard program slicing workflow—parsing the program with a Jison-generated parser; searching the parse tree for variable uses, definitions, and control blocks; computing control dependencies (e.g., dependencies from statements to surrounding if-conditions and for-loops) and data dependencies (e.g., dependencies from statements using a variable to statements that define or modify that variable); and slicing by tracing back from the slicing criteria to all the statements they depend on. When com-

puting data dependencies, we determine if methods modify their arguments by looking up this information in a custom, extensible configuration file containing data dependencies for functions from common data analysis libraries (e.g., pandas, matplotlib).

Our current implementation supports interactive computation times by splitting slicing into small, reusable parts: when a cell is executed, its code is immediately parsed, and its variable definitions and uses detected. With these precomputed pieces of analysis, gathering takes place at interactive speeds, as the most costly analyses have been performed before the analyst gathers any code.

In-lab usability study

Method

We designed a two-hour, in-lab usability study to understand the support that code gathering tools can provide to data analysts as they write code in computational notebooks. We were fairly confident of the ability of code gathering tools to eliminate clerical work—like the removal of irrelevant code, or recovery of dead code—given the design of the tool and evidence from several prior pilot studies. Therefore, the questions we sought to answer focused on the match between the control analysts desired over messy notebooks, and the support code gathering tools currently provide. We therefore designed our study to answer these research questions:

- *RQ1. What does it mean to “clean”?* When we ask analysts to clean a notebook, what do they do? Could code gathering tools support the work they are doing?
- *RQ2. How do analysts use code gathering tools during exploratory data analysis?* In our design of the tools, we hypothesized that analysts would use the tools for highlighting code, gathering to notebooks, and version browsing to find, clean, and compare versions of code. Do they?

Participants. We invited 200 randomly selected data analysts at a large, data-driven software company. The invitation stated the requirement of experience with Jupyter notebooks and Python. We recruited 12 participants altogether (aged 25–40 years, median age = 29.5 years, 3 female). Participants reported the following median years of experience on an ordinal scale: 6–10 years programming; 3–5 years programming in Python; and 1–2 years using Jupyter Notebooks. Five

participants reported using Jupyter Notebooks daily; three, weekly; one, monthly; and three, less than monthly. We compensated participants with a US\$50 Amazon gift card. In the section below, we refer to the 12 analysts from the study with the pseudonyms P1–12.

Tasks. To start, each participant signed a consent form and filled out a background questionnaire. The session then consisted of two cleaning tasks and an exploratory data analysis task. For the two cleaning tasks, we gave participants two existing notebooks from the UCSD Jupyter Notebook archive (Rule et al. 2018b), one about Titanic passengers, and one about the World Happiness Index. We chose these notebooks because they are in Python, execute without errors, use popular analysis and visualization libraries, involve non-technical domains, and are long enough to be worthy of cleaning. We counterbalanced use of the two notebooks between subjects.

For the first cleaning task, we asked the participant to scan the notebook for an interesting result and to clean the notebook with the goal of sharing that result with a colleague (10 minutes). After a brief tutorial about code gathering, we then asked the participant to repeat the cleaning task on a different notebook, this time using the code gathering features (10 minutes). Finally, for the exploratory task, we gave participants a dataset about Hollywood movies and asked them to create their own movie rankings, ready for sharing (up to 30 minutes). We chose this dataset as we thought it would be understandable and interesting to analysts from a wide variety of backgrounds. During all tasks, participants could use a web browser to search the web for programming reference material. After each of the three tasks, the participant filled out a questionnaire: the first about how they currently clean notebooks; the second about the usefulness of code gathering tools for notebook cleaning; and the third about the usefulness of code gathering tools for data exploration. Throughout the tasks, we encouraged participants to think aloud, and we transcribed their remarks.

Each participant used an eight-core, 64-bit PC with 32 GB of RAM, running Windows 10, with two side-by-side monitors with 1920×1200 pixels. One monitor displayed Jupyter Notebooks; the other displayed our tutorial and a browser opened to a search engine.

Results

The meaning of “cleaning”

Before giving analysts the tutorial about code gathering tools, we first asked them to describe their cleaning practice and to clean a notebook in their usual way. This

allowed us to understand their own interpretation of “cleaning” before biasing them with our tool’s capabilities. Many analysts explained “cleaning” in a way that is compatible with code gathering, namely keeping a desired subset of results while discarding the rest (P8, P10–12). Indeed, one analyst’s description of cleaning is surprisingly close to the code gathering algorithm: “So I picked a plot that looked interesting and that’s maybe something I would want to share with someone and then, if you think of a dependency tree of cells, sort of walked backwards, removed everything that wasn’t necessary” (P10).

In their everyday work, some analysts clean by deleting unwanted cells, but most copy/paste desired cells to a fresh notebook. (One analysts who cleans by deletion initially found the non-destructive nature of code gathering to be unintuitive, but adjusted after practice (P4).) Many described the process as error-prone and frequently re-execute the cleaned notebook to check that nothing is broken.

Every analyst reported that choosing a subset of cells is part of the cleaning process. However, for several analysts, “cleaning” includes additional activities. Several analysts reported that cleaning involves a shift in audience from oneself to other stakeholders, like peers and managers (P1, P5–7, P11). Hence, cleaning involves adding documentation (comments or markdown) (P1, P5, P7, P10, P11) and polishing visualizations (e.g., adding titles and legends) (P1, P6). Some analysts reported that cleanup includes improving both notebook quality (e.g., merging related cells (P11) and eliminating unwanted outputs (P3, P6)) and code quality (e.g., eliminating (P3, P6) or refactoring (P3, P4, P12) repeated code). Finally, for some, cleaning involves integrating the code into a team engineering process—for example, by checking the code into a repository or turning it into a reusable script (P7).

How analysts use code gathering tools to support exploratory data analysis

After both the second notebook cleaning task and the exploratory analysis task, we asked analysts to provide subjective assessments of code gathering, broken down into seven features (Figure 5.6). Gathering code to a new notebook was the clear favorite, with nearly every analyst rating it as “very useful” for both tasks. The dependency highlights were also popular. Many analysts did not find opportunities to try the version browser during the two tasks, likely due to the short duration of the lab session. Similarly, many analysts did not experience the recovery of deleted code, either because no relevant code was deleted or because the user interface recovers deleted code silently.

Valued and versatile feature of gathering code to new notebooks. Nine analysts gathered code to a new notebook at least once during the exploratory task. Analysts

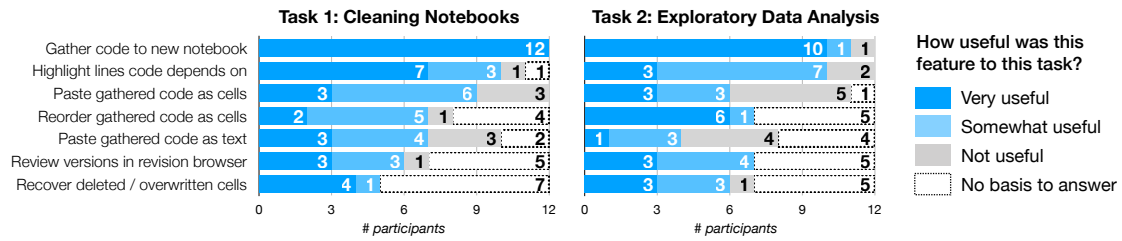


Figure 5.6: Analysts found code gathering tools most useful for gathering code to new notebooks, when they cleaned notebooks, and when they performed exploratory analysis. Analysts also appreciated dependency highlights, especially when they were cleaning code.

gathered code to a notebook a median of 1.5 times ($\sigma = 3.7$) during this task, with one analyst even gathering notebooks 12 times (P3). Analysts most often gathered code to a notebook for its intended purpose of cleaning up their code as a “finishing move” after exploration (P6). Analysts clearly valued this aspect of the tool, calling it “amazing” and “beautiful” (P10), that they “loved it” (P5), it “hits the nail on the head” (P9), and will save them “a lot of time” (P11).

Analysts saw additional value in gathering code to notebooks beyond our original design intentions. During the exploratory task, one analyst used gathering to a new notebook as a lightweight branching scheme. As he explored alternatives, he would gather his preferred alternative to a new notebook to create a clean slate for further exploration (P3). Another analyst used gathering as a way to generate reference material. She created data visualizations, then gathered them to new notebooks, so she could quickly refer back to the visualizations as she carried on exploring in her original notebook (P4). Finally, one analyst used gathering to support cleaning for multiple audiences. At the end of the exploratory task, he gathered many visualizations to one notebook and documented them for his peer data analysts; he then gathered his movie ranking result to a different notebook intended for those who only want to know the final answer (P2).

Analysts were eager to incorporate gathering into their data analysis workflows: seven of twelve analysts asked us when we would release the tool. One analyst envisioned gathering becoming part of code-cleaning parlance: “once this is public, people will send you bloated notebooks. I’ll say, nope, you should gather that” (P10).

Use of dependency highlighting. During the exploratory task, 8 analysts clicked on at least one variable definition, and 9 clicked on at least one output area. Additionally, during the cleaning tasks, as these tasks involved reading unfamiliar notebooks, a few analysts used the dependency highlights as a way to understand the unfamiliar code.

Use and disuse of the version browser. Two analysts opened the version browser at least once (P2, P3). Both copied the cells to the clipboard from a version in the version browser at least once; one analyst in fact copied cells for versions four times during their session (P2). The other analyst opened a version in a new notebook. This analyst wanted to compare versions of a cell that sorted data based on two different dimensions, and used the version browser to recover code from a prior version without overwriting the current cells, which they wished to preserve (P3).

Some analysts who did not use the version browser believed that they might eventually use it in their own work (P6, P8, P9). One analyst noted Jupyter Notebook’s implementation of “undo” is not sufficient for them, and the version browser could provide some of the backtracking functionality they want (P6). Another reported that the version browser could be useful in their current work, where they have iteratively developed an algorithm and are managing three notebooks containing different versions of analyses (P9). However, two analysts believed they wouldn’t use the version browser, as its view of versions is too restrictive. The version browser collects versions ending with multiple executions of the same cell, yet these analysts preferred to modify and re-run old analyses in new cells (P10, P11).

Downsides and gaps. A few analysts mentioned that repeatedly gathering code to a new notebook creates a different kind of mess, namely clutter across notebooks, rather than clutter within a notebook. For example, gathering multiple times typically causes initialization code (e.g., loading the dataset) to be duplicated in each generated notebook (P3, P4, P6). In effect, a notebook and the notebooks gathered from it form a parent/child relationship that the user interface does not currently recognize. Analysts suggested several improvements. First, gathering to a new notebook should create a provisional notebook, rather than being saved by default, and its name should be related to the original notebook’s name. One analyst suggested linked editing across this family of notebooks as a way to deal with duplicated code. For example, renaming a variable in one family member could automatically rename it in all members (P12).

Two analysts believed that comments, when close to the code, should be gathered alongside the code they comment on (P3, P10). One of these analysts noted that including irrelevant comments would not be problematic, as “it’s easy to remove some extraneous text” (P10).

Validating the design motivations

Analysts’ feedback offered evidence of the role that our design motivations played in the usefulness of the tools:

Post-hoc management of messes. Analysts valued the ability to manage messes without up-front effort to organize and version code. This was a benefit of gathering to new notebooks, as analysts appreciated simple affordances to clean up their messy analysis code (P1, P2, P6, P8, P9). For one analyst, the tool encouraged them, for better or worse, to “not to care... too much about data cleaning or structure at this moment. I say it was nice in a way, that I can just kind of go on with what I want to do” (P12). For some analysts, this was the downside of the version browser, which required them to run new versions of code in the same cell (P10, P11).

Portability of gathered code. Analysts reused gathered code by opening fresh notebooks, pasting cells, and pasting plaintext. In the exploratory task, nine analysts gathered code to notebooks, and five gathered code to the clipboard, to paste as either cells or plaintext. By pasting plaintext into one cell, analysis code looked “a lot cleaner” (P3), and several analysts wanted an easier way to gather code to scripts. Others preferred pasting code as distinct cells (P5, P7). One analyst simply liked having the choice (P10).

Querying code via direct selection of analysis results. Analysts appreciated the directness with which they could gather code: “It was very easy to just click, click, click on something and then grab the code that produced” a result (P10). The directness allowed analysts to clean their code by asking, “what do I need?” rather than “what do I not need?” (P3).

Conclusions

Our qualitative usability study with 12 professional data scientists confirmed that cleaning computational notebooks is primarily about removing unwanted analysis code and results. The cleaning task can also involve secondary steps like improving code quality, writing documentation, polishing results for a new audience, or creating scripts. Participants find the primary cleaning task to be clerical and error-prone. They therefore responded positively to the code gathering tools, which automatically produce the minimal code necessary to replicate a chosen set of analysis results, using a novel application of program slicing. Analysts primarily used code gathering as a “finishing move” to share work, but also found unanticipated uses like generating reference material, creating lightweight branches in their code, and creating summaries for multiple audiences.

Limitations and extensions

Our study has two limits to external validity, which are common in lab-based usability evaluations: first, the participants did not do their own work, on their own data, in their own time frame. We created realistic tasks by choosing notebooks and datasets from the UCSD Notebook Archive, itself mined from GitHub. Ideally, participants would use their own data and analyses. However, several informants in our formative interviews said their data was too sensitive for us to observe, so we did not pursue this option. The second limitation is the study’s short duration, which we believe accounts for the low use of the version browser feature. As P6 commented, “the other features will be more valued for notebooks that have been used for a long time/long project.”

To help analysts manage messes in their code, we offer tool builders the following suggestions:

Support a broad set of notebook cleaning tasks. While slicing and ordering code is a key step in cleaning notebooks, analysts still need support for many other cleaning tasks. This includes refactoring code (e.g., eliminating duplicates and extracting methods), restructuring notebooks (e.g., merging cells), polishing visualizations, and providing additional documentation to explain the code and results. Many of these tasks still lack tool support in computational notebooks.

Design versioning tools to support many ways of organizing code. In our study, and in Rule’s (2018) study of Janus, analysts used cell version histories less than expected. Is this because of issues in tool design, or because of the studies’ length? Evidence from our study lends credence to both claims. Some analysts in our study told us they would not use the “Gather to Revisions” feature, as they wrote versions of code in a way that our system could not detect, i.e., duplicating and changing a cell’s content elsewhere in the notebook. For future tools, two cells’ “sameness” should not be determined by a cell’s placement, but perhaps by using heuristics such as text similarity. Furthermore, several participants reported they didn’t have enough time to create versions during the study, suggesting the need for longer programming sessions, and perhaps long-term deployments, in future studies.

Code gathering with history and compositionality. The code gathering tools’ execution log lasts only for a single programming session, which limits the scope of the Revisions button and resurrecting code from deleted or overwritten cells. Future tools should use a persistent execution log. Several participants wanted to create a cleaned notebook in a series of steps, that is, for a gathering step to “patch” notebooks gathered in a previous step. Future tools could use algorithms from revision control systems to support this flexibility.

Reuse code gathering tools in other programming environments. Code gathering can be useful in other tools, such as read-eval-print loops for interpreted languages like R, Python, Scala, and others. These interpreted languages are another popular category of tools for data analysts.

Chapter 6. Tutorial distillation

Flexible sequencing of snippets

“Great things are not done by impulse, but by a series of small things brought together.”

Vincent Van Gogh

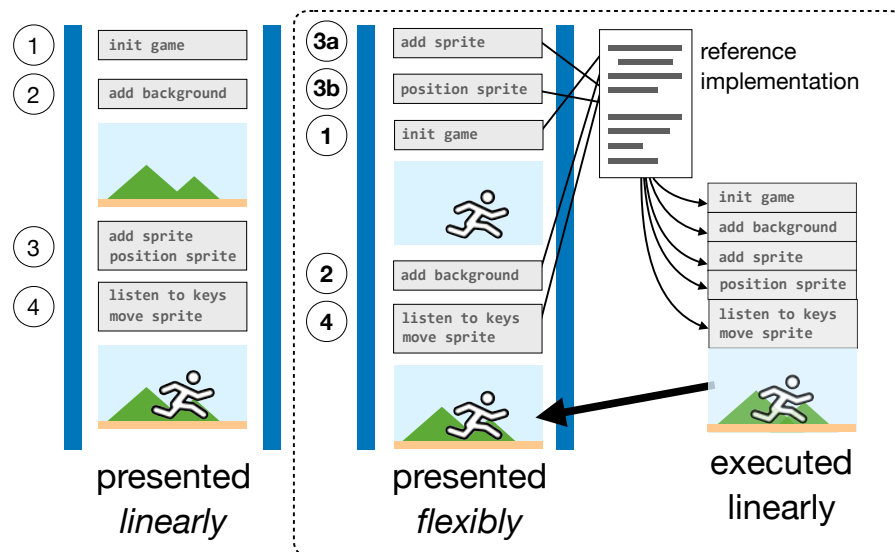


Figure 6.1: Interactive tools for creating tutorials typically support linear presentation of code, though authors often present code with repetitions and fragments. We propose a tool called Torii that enables the creation of tutorials with flexible presentation of code snippets while keeping code and outputs consistent. The tool (shown within dotted lines above), preserves links from snippets to a reference implementation to preserve consistency, and to determine how outputs should be generated from snippets.

Motivation

In 1984, Donald Knuth proposed *literate programming* as a new approach to writing code. In this vision, instead of programs, authors write about computational ideas and the implementation of those ideas. Instead of simply commenting their source code, a programmer splits their program into brief code snippets, and interleaves these snippets with explanations about what the snippets do, and how they fit together into a complete program. The output of literate programming is a document that describes an algorithm, studded with code that shows how each piece of the algorithm is implemented (Knuth 1984).

Today, the vision of literate programming has become manifest in the form of tutorials that programmers write for one another. Bloggers (Parnin et al. 2013), open source developers (Dagenais and Robillard 2010), and technical writers all create and share tutorials on the web. Sites like Ray Wenderlich host thousands of tutorials written by hundreds of authors. Companies like Apple produce hundreds of tutorials to help programmers use their development tools (Tiarks and Maalej 2014). These tutorials go beyond textual presentation to include visuals (screenshots, videos), and interactive components (running programs, embedded demos that update with new output as a reader edits a code snippet).

While literate programming has become the pervasive paradigm for tutorials about programming, the tools that authors use to produce these documents have not seen a similar renaissance. Instead, tutorial authors typically use text editors for the prose and code portions, and standalone tools for running code and producing images and videos. One notable exception is the interactive computational notebook, which has become popular for many programming tasks, including authoring tutorials in domains like data analysis.

However, there are many programming tasks for which the notebook paradigm is insufficient. These include user interface development, web server implementation, game development, and visualization creation. For this kind of programming, code may not be readily presented in an order that can be interpreted or compiled. Rather, it is best explained as an incremental refinement to a base program. For this kind of program, tutorial authors continue to depend on general purpose text editors rather than computational notebooks.

To advance the state of the art in tutorial authoring tools, this chapter first describes the special challenges of the programming tutorial authoring process and then presents and assesses a prototype tool with novel features for enabling flexible presentation of code, and keeping snippets consistent with outputs.

To understand the key needs for tutorial authoring, we conducted two different

qualitative studies. One was an in-depth interview study with 12 accomplished tutorial authors, which found that, compared to other online content creators, tutorial authors faced a unique challenge of keeping collections of related programming artifacts consistent with each other as they wrote and revised a tutorial. In essence, writing a tutorial often entailed creating several artifacts in parallel—a source program, the snippets derived from that source program, prose explanations of the snippets, and outputs generated from the source program. Authors were sometimes dissatisfied with their tools and processes for keeping these artifacts consistent. A secondary issue was the desire for more support for producing “assets”: outputs generated by running code snippets, diagrams, screenshots, and demos.

To verify that the problems identified were representative of popular tutorials, we report on a content analysis of 200 widely-referenced web-based programming tutorials. A majority included code fragments that showed only a portion of a source file (83%). Many included assets such as screenshots, diagrams, videos, and embedded demos of running the code (80%). Most tutorials also included resources that would need to be kept consistent with each other should the tutorial be further changed, such as duplicated code (59%) and outputs generated from running the source program (67%).

To understand how tools can help authors write tutorials, we designed, implemented, and assessed a prototype tool called Torii.¹ This tool helps authors keep their source programs, snippets, and generated outputs consistent with one another, and allows the author to organize and present code in the order they see fit (Figure 6.1). This includes showing the same code in multiple locations, from different points of view, explaining code snippets out of their original source code order, and showing code snippets that are syntactically invalid in isolation, but valid when combined with other code in the tutorial. In other words, Torii helps authors *supplement* the code in a tutorial with outputs, while providing support for flexible *sequencing* of the code. We assessed this tool in an in-lab study with 12 participants, finding positive usability outcomes for many of the proposed features, and directions for improvement for others.



¹ **Torii** (tOR-ee-ee) *n.* **1** A gate marking passage from the mundane to the spiritual. **2** An abbreviation of the word “tutorial.” **3** A tool, described in this chapter, that propagates changes between source programs, snippets, and outputs in a tutorial workspace.

Formative study I: Interviews with tutorial authors

Building on past content analyses (Kim and Ko 2017; Nasehi et al. 2012; Tiarks and Maalej 2014) and qualitative studies of the authoring process (Mysore and Guo 2017; Parnin et al. 2013), we conducted two studies to expand our understanding of tutorial authoring. In our first study, we interviewed 12 authors to develop a rich, qualitative understanding of how programming tutorials are constructed.

Method

Participants. We contacted recently active authors from a sample of online programming blogs. Of the approximately 50 authors we emailed, 12 opted to participate (referred to as A1–12 below). Recruited authors had considerable experience. Each had written from a few to over one-hundred tutorials. Authors lived in at least four different countries, and consisted of both amateurs and paid professional technical writers.

Interviews. Interviews were semi-structured and lasted between 30 minutes and an hour long, with one interview scheduled for an additional one-hour follow-up. Authors were asked to describe how they wrote tutorials, the challenges they faced, and how they thought tools could help them write tutorials better. Audio was recorded for all interviews, and anonymized transcripts were made for each interview.

Analysis. One researcher analyzed the interview data, following a qualitative approach described in Weiss' seminal guide to conducting interview studies (Weiss 1995). Throughout the analysis process, themes were refined, hypotheses developed, and relevant passages excerpted.

Results

Overview

Authoring a tutorial is an effort-intensive process that involves picking ideas to write about, building prototypes, testing out the code, writing excellent prose, and disseminating the work. Interviewees described, in each of these stages, the challenges they faced: finding topics that are sufficiently unique to write about (A3, A6), finding high-quality copy-editors (A5, A8), and producing content on a regular cadence (A8, A10). In reporting these results, we highlight only the

authoring challenges unique to programming tutorials, with an emphasis on the production and presentation of code.

Keeping source code, snippets, and outputs consistent

As an author writes a tutorial, they are in essence developing and maintaining four types of resources in parallel:

A source program or a set of source programs that they are trying to describe to a reader, or teach a reader to build.

Snippets of code taken from these programs from a specific point of time in the development of those programs. The snippets are embedded in the tutorial as focused and often short views of the source programs.

Prose explanations of snippets and how they fit together into a program, and of algorithms, concepts, and anecdotes germane to the tutorial’s narrative. Diagrams may augment the prose.

Outputs produced by running selections of code from the source program. These include console logs, user interface screenshots, and embedded, running demos (e.g., web pages embedded in `iframes`, interactive visualizations).

While these resources are distinct artifacts in the author’s workspace, many of them are different views of the exact same code (Figure 6.2). Snippets often represent a partial view of a source program at one point in its development. Outputs are generated from running a version of the source program, some of the code for which may appear in the snippets. Meanwhile, snippets themselves may appear more than once in a tutorial, or part of the code of one snippet may appear in another.

These relationships are not recorded by the tools authors used to make tutorials. One of the most common annoyances authors described was simply keeping all of these resources in sync. Because the contents of these resources are so closely related to each other, interviewees reported needing to perform several tedious and error-prone tasks to keep their programs, snippets, and outputs consistent with each other:

Starting with a reference implementation. Some interviewees built a complete reference implementation before adding code to a tutorial. Three professional authors for the same online tutorial portal (A9, A11, A12) were required to produce a “starter project” and a “final project,” and have this code checked off before they began to write a tutorial. Another tutorial author started writing complete im-

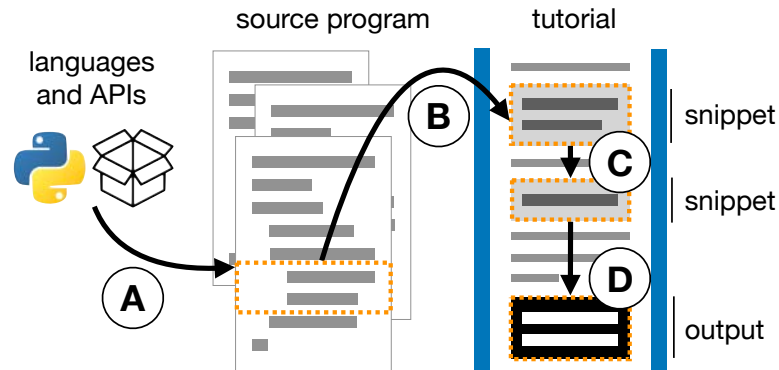


Figure 6.2: When writing a tutorial, authors clone and transform code in ways that are not tracked by conventional development tools. Source programs depend on languages and APIs, and may need to be updated when these change (A). Snippets are copied from the source program into snippets (B), and the same code may appear in multiple snippets (C). Outputs are generated by assembling and executing the snippets (D).

plementations after an experience where they found they had painted themselves into a corner and needed to change their approach mid-tutorial (A8).

Propagating code changes. When an author changes a snippet or a source program, they must make sure that the change is reflected in all other versions of the source program and all other snippets. Interviewees reported needing to propagate changes like these for both larger tutorials and for books (A2, A5, A9, A11). These code changes could be triggered by forces outside of the author’s control, like changes to the APIs and frameworks used by the tutorial’s code (A5).

Play-testing the tutorial. If an author plans to publish a completed source program for a reader’s reference, they need to make sure that the reader, after assembling all of the snippets in the tutorial, will end up with the same code as the published source program. One author followed along with their own tutorials, checking to see that they finished with the same code as the reference program they wanted to post (A12).

Regenerating program outputs. When an author changes the code in a snippet, they must change the outputs that depend on that snippet, which may be numerous. In one author’s case, these outputs were screenshots of a running interface (A6).

Authors adopted strategies to overcome this brittleness in the tutorial authoring workspace. They architected code to minimize dependencies (A4), backed the source program with a version repository so changes could be readily propagated across versions of the source program (A1, A5, A9, A11, A12), and embedded version-controlled snippets in the tutorial (A1). No interviewees had workarounds

to easily update snippets or outputs when changes were made to the source program.

Presenting code and outputs

Authors wanted their tutorials to be engaging, easy-to-read, and informative. All authors were deeply concerned with readers' expectations and the experience they would have reading the tutorial. They designed, and revised, tutorials to ensure they could hold a reader's attention, and that the target reader could successfully follow the tutorial. This concern for the reader's experience manifested in common design choices for presenting code, outputs, and other visuals.

Keeping code minimal. Authors were aware that the code snippets in a tutorial could be one of the most cognitively demanding parts of the tutorial for readers to engage with. Most authors were minimalists when it came to code, showing no more code than was necessary (A2, A3), simplifying code until it became easy to explain (A11), and keeping snippets short. Authors scoped snippets to small, self-contained units of functionality (e.g., individual functions) (A4, A11) and, if code was sufficiently complex, introduced code just one line at a time (A1). Authors highlighted important spans of code by styling the code (A2, A10), or adding numeric labels to the comments that they referred to from the prose (A11).

“Breaking up the text.” Authors sought to keep text brief and clear. “Walls of text” were to be avoided and split up. One interviewee, for instance, told us he tried not to write tutorials longer than 500 words (A1). Code, quotes, and screenshots served dual purposes of both conveying important information, and breaking up the text (A1, A2, A6).

Integration of videos, diagrams, and memes. With only text and code, a tutorial might be dry, or inefficient at explaining key concepts. Authors incorporated several types of “assets” into tutorials to make them more engaging and to more effectively convey key concepts. They injected humor and encouragement into their tutorials by adding topical memes and icons (A2, A4). Authors sometimes felt it was more appropriate or effective to convey ideas with videos (A5, A12) or diagrams (A8, A11) than with text and code alone. Screenshots could be introduced to help readers check their work (A11). However, assets like videos and diagrams could take quite a bit of effort to design and produce (A8, A11).

A desire for interactive outputs. Only one author included interactive affordances in her tutorials, wherein readers could tinker with code in interactive editors and see program outputs change live (A6). Several authors wanted to include interactivity

in their tutorials (A8, A11, A12), believing it could help readers better understand the code.

Formative study II: Content analysis of two-hundred tutorials

To verify that the pain points identified in the interviews were representative, we performed a content analysis on a representative set of web-based programming tutorials. This content analysis provides context for tool design, revealing the prevalence of “flexible” code organization and generated outputs in tutorials. Compared to an automated analysis, a content analysis let us detect the presence of code fragments, duplicated code, and generated outputs, which are tricky to identify without human inspection.

Method

Selection. To identify a diverse sample of popular (and therefore presumably high-quality) tutorials, we searched Stack Overflow answers for external links with the anchor text “this tutorial.” This yielded over 20k candidate links to tutorials. We filtered these links to those that appeared in an answer with one or more up-votes, and with two-hundred or more referring domains, as determined using a backlink service. We randomly sampled the remaining tutorials until we had a set of 200 tutorials, omitting those which on inspection lacked prose or contained fewer than two related code snippets.

Analysis. Two authors independently analyzed and labeled the tutorials with 23 variables, including the number of code snippets, presence of fragmented code snippets, and presence of generated outputs.² This analysis resulted in substantial agreement for all variables on the first pass (Krippendorff (2013) $\alpha = 0.75 - 0.98$). The authors reviewed their labels for errors with reference to each other’s labels (attaining $\alpha = 0.93 - 1.0$), and settled all remaining disagreements together.

Results

Overview. Tutorials ranged from extremely brief—four tutorials with only two snippets—to extremely long—five tutorials with more than 100 snippets. The median tutorial contained 11 snippets, though tutorials varied widely in their number

² A complete listing and codebook appears in the auxiliary material distributed with the published conference paper (Head et al. 2020).

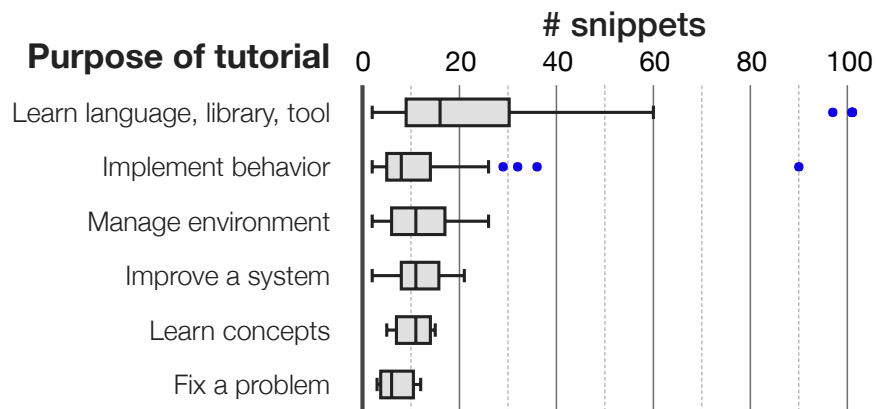


Figure 6.3: The typical tutorial contains 11 snippets—though this number varies depending on the tutorial’s purpose. Tutorials about learning a language, tool, or library had far more snippets than those about implementing a behavior. The box plots above show the distributions of snippet counts for each type of tutorial; blue dots are outliers.

of snippets ($\sigma = 18.9$), with a long right tail (Figure 6.3). A summary of the analysis results is shown in Table 6.1.

Each tutorial was assigned one of six primary learning goals. The most common goals were to learn about a language, library, or tool (43%), and to implement a behavior (40%). Far less common were tutorials focusing on helping readers manage their development environment (11%), improve an existing system (4%), learn abstract programming concepts (2%), or fix a programming problem (2%).

Fragmented code snippets. 83% of tutorials included at least one *fragment*, which we defined as a piece of code the reader should place in a file, but which was not intended to stand on its own. Often, fragments would not be able to be compiled or interpreted until a reader integrated it with additional code. Sometimes fragments were the result of authors hiding code that was shown in an earlier snippet.

Code duplication. In most tutorials (59%), code from one snippet was reused in another snippet. In many cases, the repeated code served as context to show where new code was being added, or other code was being updated. Other times, a fragment of code was pulled from an earlier snippet to show on its own. In 48% of tutorials, code from one snippet was changed, partially or wholesale, in a later snippet.

Generated outputs. Most tutorials contained outputs generated by running some of the tutorial’s code (67%). The two most common types of generated outputs were console logs (33%) and images (e.g., screenshots of running applications, 32%).


































Purpose of tutorial	All tutorials	Learn language, library, tool	Implement behavior
# Tutorials	200	85	79
Fragments	83% 	84% 	91% 
Duplicated Code	59% 	64% 	62% 
Rewritten Code	48% 	56% 	44% 
<i>Any Generated Output</i>	67% 	61% 	77% 
Console Output	33% 	38% 	20% 
Images of Output	32% 	24% 	46% 
Videos of Output	6% 	2% 	11% 
Text File Output	4% 	2% 	4% 
Linked Demo	15% 	16% 	19% 
Editable Demo Code	5% 	8% 	3% 
Other Visuals	55% 	49% 	62% 

Table 6.1: Programming tutorials often contain code fragments, duplicated code, and generated outputs. Shown are percentages of tutorials with code fragments, duplicated code, and eight other characteristics. Percentages are shown for two major categories of tutorials—learning a language, library, or tool; and implementing a behavior—and for the dataset as a whole.

Tutorials occasionally contained live demos of running code within the page itself, or at an easily accessible link (15%). In rare cases, code for these demos could even be edited and re-run (5%). Other types of generated outputs included videos (e.g., screencasts of running the code, 6%) and text files generated by running the source program (4%).

Other assets. Most (55%) of tutorials contained non-output visuals, like diagrams (24%), user interface screenshots (21%), or other images (e.g., logos, ads, 33%).

Style. 10% of tutorials applied special styling to notable code in at least one snippet, and 7% applied special styling to indicate what changed in a snippet versus an earlier snippet. 44% added placeholders (Buse and Weimer 2012) to snippets to show where readers should supply their own code or fill in code in a later step. 13% contained snippets with “cuts,” or explicit markers (e.g., “...”) to indicate that code from an earlier snippet was hidden. 5% included numerical or textual labels in the code (e.g., “// 1,” “// 2”) referenced from the tutorial’s prose.

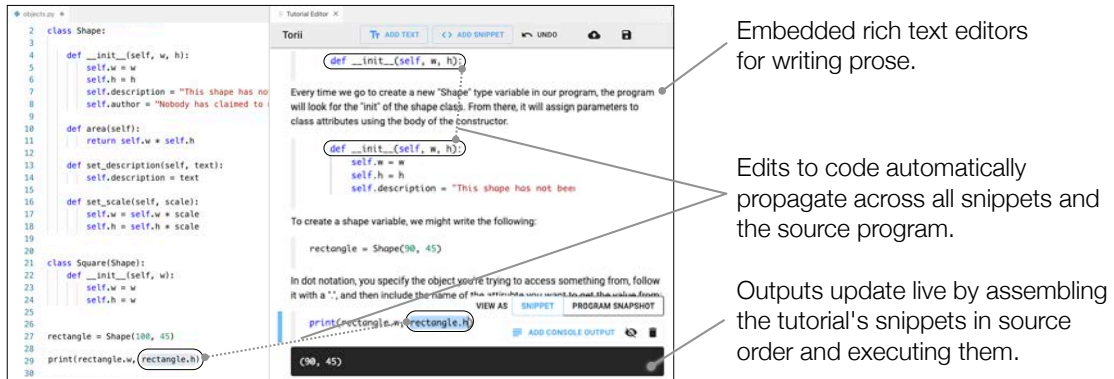


Figure 6.4: Writing tutorials with Torii. Torii helps authors write tutorials by keeping source programs, snippets, and outputs consistent with each other, while still letting authors organize the code in the tutorial flexibly. An edit to code anywhere in the tutorial workspace automatically triggers an update to clones of that code in the source program and snippets, and to all outputs generated from that code.

A demo of Torii

Informed by our formative research, we designed Torii as a prototype tool to help authors create programming tutorials. The design was motivated by two goals:

1. **Consistency.** Help authors keep source programs, snippets, and generated outputs consistent with each other.
2. **Flexibility.** Provide authors freedom to present code—that is, to split, order, and repeat it—as they see fit.

To provide a consistent and flexible authoring workspace, at the beginning of tutorial creation, Torii takes as input a reference implementation of the source program. Authors create code snippets as partial, editable views of the reference implementation. Outputs are generated by assembling snippets in the order they appear in the reference implementation. Our interviews found that many authors have such a reference implementation available when they start writing a tutorial.

To demonstrate the experience of authoring tutorials with Torii, we describe how a hypothetical author, Rhia, writes a tutorial about the basics of object-oriented programming in Python.³ Rhia wishes to present code with a level of flexibility she cannot achieve with other literate programming interfaces like notebooks. For example, Rhia wants to split classes into short snippets that can be

³ The code for the tutorial in this scenario is adapted from the “Classes” chapter of *A Beginner’s Python Tutorial* (Wikibooks contributors 2019), published under the Creative Commons Attribution-ShareAlike 3.0 license

explained in isolation, but which would not compile if executed separately. In the scenario below, descriptions of Torii’s key affordances are interspersed with screenshots and implementation details for each affordance.⁴

Propagating edits from snippets to source programs

Rhia invokes a command to launch Torii in her integrated development environment. This brings up a pane containing a WYSIWYG tutorial editor (Figure 6.4). To add the first snippet to her blank tutorial, Rhia selects a few lines of code in the source program’s code editor, and then clicks the “Add Snippet” button in the tutorial editor. Torii wraps the selected code in an embedded code editor and places it as a “snippet” in the tutorial editor. The snippet is directly editable and linked to the source program: any change to the snippet propagates immediately to the source program, and vice versa.



Implementation. Torii maintains a map between each snippet and the location (i.e., line numbers) it was copied from in the source program. When an author edits code, Torii detects where the edited code appears in other snippets and the source program, and translates the edit action into edit commands to be dispatched to each snippet and source program editor.

Propagating edits from code to outputs

Once Rhia inserts several snippets and descriptions of those snippets, she adds an output to demonstrate what the program is doing. Rhia inserts a snippet containing a `print` statement, and clicks the “Add Console Output” button that appears directly below the snippet. Torii generates an output by running the snippets above it, and inserts it into the tutorial.

⁴ See also the demo video distributed with the published conference paper (Head et al. 2020).



The output is linked to the code in the workspace. As Rhia tinkers with the source program or the snippets in the tutorial above it—e.g., to change the initialization parameters of an object, or to change a method body—the output updates automatically to reflect the changed code.

Splitting, reordering, and copying code

Rhia splits and organizes the source program into snippets in just the way she wants. Torii lets her split it into syntactically-incomplete snippets if she pleases. It also lets her hide snippets that contain boilerplate (e.g., `import` statements) necessary for generating an output, but which might be distracting to a reader. As long as all necessary code appears in a snippet above an output, Torii figures out how to assemble the snippets to generate and update the outputs.

In this case, Rhia takes advantage of the flexibility Torii provides to show the usage of a class before its declaration, and to show individual methods and properties of a class outside of the class declaration. Rhia also repeats the same code twice in two snippets, showing the same line once in the context of a method definition, and then again on its own with a detailed explanation. Torii correctly infers that the duplicated line should only be run once when generating the outputs.

out-of-order declarations

```
rectangle = Shape(90, 45)
```

• • •

```
class Shape:
    def __init__(self, w, h):
        self.w = w
        self.h = h
        self.description = "This sha
        self.author = "Nobody has cl
```

split structures

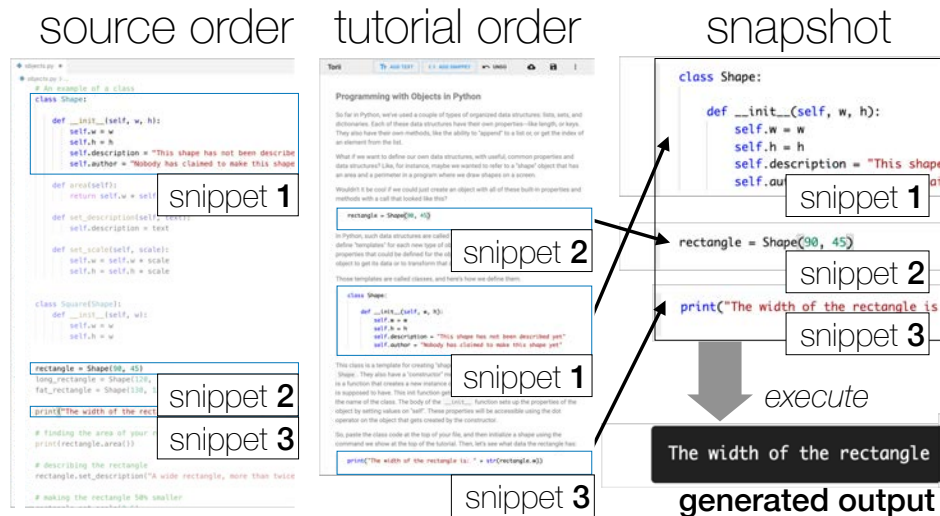
```
class Shape:
    def __init__(self, w, h):
        self.w = w
        self.h = h
        self.description = "Thi
        self.author = "Nobody h
```

• • •

```
def area(self):
    return self.w * self.h
```

Implementation. Because Torii remembers snippets’ locations in a source program, it can infer how to “stack” snippets correctly into executable programs. For each output in a tutorial, Torii assembles a *program snapshot*: an executable program comprised of all snippets—in order and deduplicated—that appeared above the output element in the tutorial.

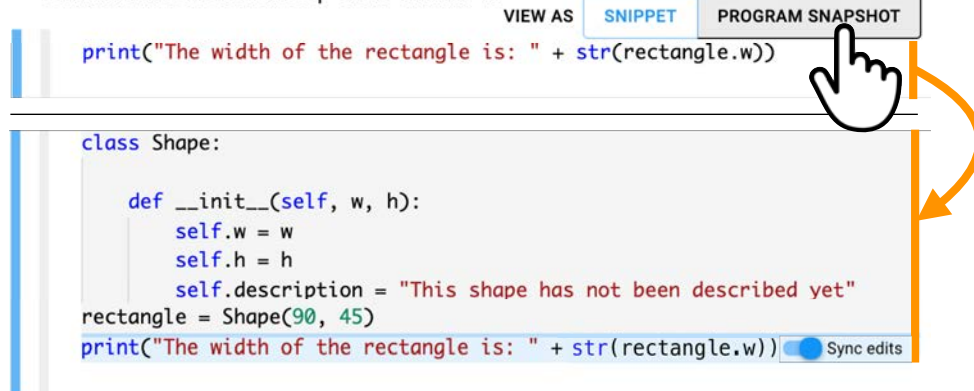
To build a snapshot, Torii takes all snippets that appear above the output (including hidden snippets), orders them by their location in the source program, and removes duplicated lines. To generate an output from the snapshot, the snapshot is written to temporary files, and executed using a configurable code runtime—in this case, the Python 3 command. The output of the runtime is piped into the output element in the tutorial:



Reviewing a simulated reader’s code

Rhia can click on the “Program Snapshot” tab in any snippet to see what Torii would execute to produce an output at that point in the tutorial. Most practically, this snapshot provides Rhia a view of the code the reader will have at this point in the tutorial, if they assemble the snippets in the order they appeared in Rhia’s reference implementation.

So, paste the class code at the top of your file, and then initialize a shape using the command we show at the top of the tutorial. Then, let's see what data the rectangle has:



VIEW AS SNIPPET PROGRAM SNAPSHOT

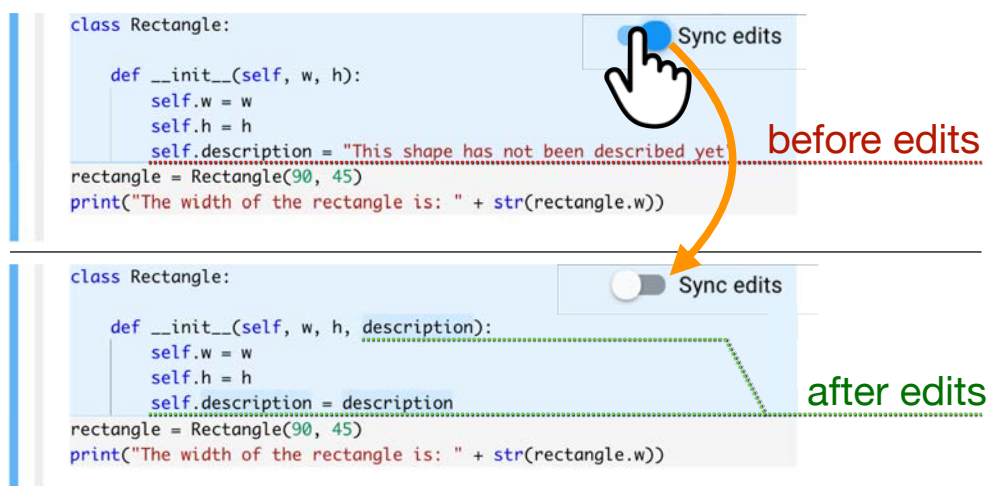
```
print("The width of the rectangle is: " + str(rectangle.w))
```

```
class Shape:
    def __init__(self, w, h):
        self.w = w
        self.h = h
        self.description = "This shape has not been described yet"
rectangle = Shape(90, 45)
print("The width of the rectangle is: " + str(rectangle.w))
```

Sync edits

Making localized changes to the code

Rhia adds a step to the tutorial that requires readers to change a method signature from an earlier snippet. Torii helps her do this by letting her make an edited copy of the snippet. All snippets below the copy will have the changes, and all snippets and outputs above will be left untouched. To make this edited copy, Rhia adds a snippet containing the method a second time. She then turns off synchronization between this snippet and prior snippets by clicking on the “Sync edits” toggle button, which can be found in the snapshot preview for the snippet.



class Rectangle:

```
def __init__(self, w, h):
    self.w = w
    self.h = h
    self.description = "This shape has not been described yet"
rectangle = Rectangle(90, 45)
print("The width of the rectangle is: " + str(rectangle.w))
```

before edits

class Rectangle:

```
def __init__(self, w, h, description):
    self.w = w
    self.h = h
    self.description = description
rectangle = Rectangle(90, 45)
print("The width of the rectangle is: " + str(rectangle.w))
```

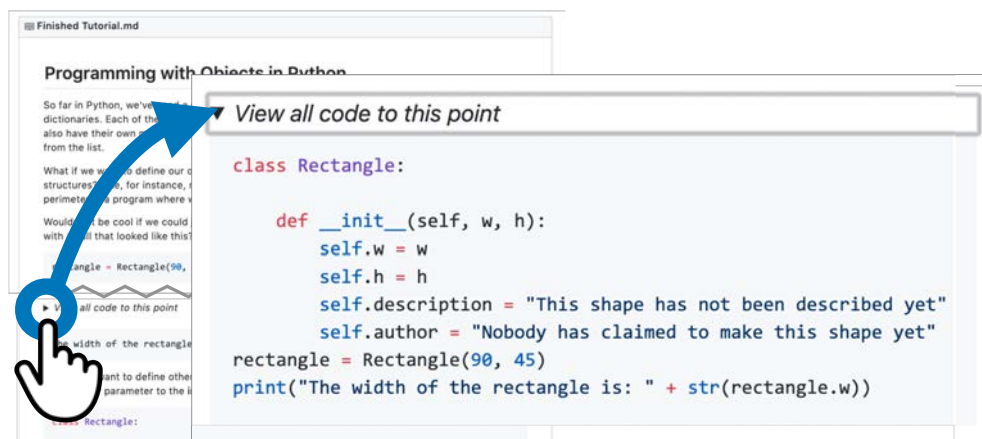
after edits

Implementation. When Rhia disables edit synchronization for a snippet, Torii creates a fork of the snippet with the same code and breaks the fork’s link to prior

snippets. When generating outputs, Torii builds a program snapshot to include only the last version of the snippet that appears above the output. The current design of localized changes was chosen to resemble the linked editing interaction technique (Toomim et al. 2004), which was designed to support simultaneous edits of partial code clones.

Distributing augmented tutorials

Once Rhia finishes the tutorial, she uses Torii to save it as an augmented Markdown document. The document includes all richly-formatted text, snippets, and outputs she created in Torii. In addition, Torii exports snapshots after each snippet, placing them behind expandable headers, which readers can toggle open to check their work.



In-lab usability study

We designed an in-lab usability study to provide an initial assessment of Torii as a tutorial authoring tool. Can authors use a tool like Torii to create and update programming tutorials? Do they leverage its unique execution model to create tutorials that wouldn't be possible in existing tools like notebooks? This study yielded insight to guide the design of future versions of this tool and other authoring tools.

Method

Recruiting. We invited local tutorial authors to participate in a 1.5-hour lab study. To reach these authors, we sent invitations to one Facebook page, one Slack channel, and one email list, each reaching a different group of local computer science and programming educators. Candidates were screened for experience writing at least one programming tutorial, and for comfort with the Python programming language. Authors were recruited from among local educators with tutorial-authoring experience, rather than remote experts, to allow for a controlled study appropriate for assessing a prototype.

Participants. 12 authors were recruited. We refer to these participants as P1–P12 below. All participants had previously written a programming tutorial, and all had experience creating other instructional materials (e.g., programming lectures, lab guides). Several participants had considerable experience—one wrote a textbook (P12), one wrote tutorials for open source libraries they maintained (P4), and another created on-boarding materials in industry (P8). Among participants were six undergraduate students, three graduate students, one professor, one software developer, and one data scientist. All participants had at least 1 year of Python programming experience, and the median participant had 3–5 years of experience.

Procedure. The study consisted of training, two tutorial maintenance tasks (with three subtasks each), and an open-ended tutorial authoring task. At the study’s conclusion, participants were compensated with \$30 gift cards.

Training. To learn how to use Torii, participants followed along with a guided tool walkthrough. The walkthrough guided participants in embellishing and editing an existing tutorial. By following along, a participant used all of Torii’s features, except for features for saving the tutorial. The tutorial that participants edited was based on the Tic-Tac-Toe tutorial from *Automate the Boring Stuff with Python* (Sweigart 2020).

Because code execution in Torii worked differently than in most programming environments, participants were encouraged to ask questions and check their understanding with the experimenters. This phase of the study took 15–40 minutes, depending on each participant’s pace, and how long they believed they needed to understand the tool.

Maintenance tasks. Then, participants completed two tutorial maintenance tasks. One task was completed with Torii. The other was completed with a comparison tool: *VSCode*, augmented with a plugin for editing and rendering Markdown files (*Markdown Preview Enhanced*). In the comparison condition, participants

had access to Markdown syntax highlighting, live rendering of the Markdown tutorial, and a built-in terminal for running code.

Each maintenance task comprised three subtasks:

- (a) *Linked edit*: Change a literal value, and update the text and outputs to reflect the new value.
- (b) *Localized edit*: Make a change to a function argument that is localized to one part of the tutorial.
- (c) *Revert edit*: Revert the localized edit made in subtask *b* in another snippet, later in the tutorial.

Subtask *a* represented routine edits authors make to keep tutorials consistent, a need uncovered in the interviews. Subtasks *b* and *c* were designed to measure performance with Torii’s specific features for localized changes.

Before a task, participants were given up to five minutes to review the tutorial and the source program it was based on. For the next ten minutes, they completed as many subtasks as they could, in order. For each task, they were assigned one of two different tutorials. Both tutorials were based on chapters in DigitalOcean’s *How to Code in Python 3* guide (Tagliaferri 2020). They contained about the same number of lines of code, with approximately the same code complexity. The order of tutorials and tasks was counterbalanced between participants.

Authoring task. In the remaining time (15–30 minutes, depending on participant), participants completed an open-ended authoring task. This task let us observe how authors would use Torii’s affordances for flexible code organization when creating a tutorial from scratch with a source program. Participants were asked to create a tutorial explaining the basics of object-oriented programming in Python. They were given a source program demonstrating basic object-oriented programming operations, derived from the “Classes” chapter of the *A Beginner’s Python Tutorial* Wikibook (Wikibooks contributors 2019). Participants were encouraged to keep the tutorial’s prose simple so they could spend more time with the tool’s affordances for organizing code. Modifications to the source program were permitted.

Questionnaires. Participants filled out four questionnaires: one following each maintenance task (both conditions), one more after the last maintenance task, and one after the open-ended authoring task.⁵ Study sessions concluded with brief oral

⁵ Due to technical difficulties, a handful of questionnaires and timing data are missing. The first three questionnaires and maintenance task times for two participants (P1, P2) and the final questionnaire for one participant (P11) are omitted from analysis.

question and answer periods in which we asked participants to reflect on their experience using Torii.

Results

Maintenance and creation of tutorials

Maintenance tasks. With Torii, participants completed most tasks—10 of 10 finished subtask *a*, 9 finished subtask *b*, and 3 finished subtask *c*. Participants achieved similar completion rates with the control interface: 10 of 10 finished subtask *a*, 5 finished subtask *b*, and 7 finished subtask *c*.

Low completion rates for subtask *c* can be interpreted as an opportunity to improve Torii’s design. Most (6 of 7) participants who failed to complete subtask *c* shared a misconception: that to revert a localized change, they only needed to copy a snippet once more from the source program with the original code. The prototype of Torii required an additional step of “unsyncing” the copied code, though in retrospect we believe this design is neither intuitive nor ideal.

Participants reported completing subtask *a* and subtask *c* more quickly with Torii, and subtask *b* more quickly with the comparison interface. With Torii, subtask *a* was finished in a median of 45 seconds ($\sigma = 32s$) rather than 88 seconds ($\sigma = 86s$), and subtask *c* in a median of 57 seconds ($\sigma = 33s$) rather than 67 seconds ($\sigma = 46s$).

Subtask *b* appeared to take quite a bit more time with Torii than the comparison tool. Using Torii, participants reported completion in a median of 3 minutes and 47 seconds ($\sigma = 2m\ 2s$) rather than 2 minutes and 20 seconds ($\sigma = 54s$). This timing difference would suggest that the localized edit functionality is perhaps unintuitive, and that this affordance of the system could benefit from further design.

These differences in task times between conditions are, we note, not statistically significant with a Wilcoxon two-tailed signed-rank test. This is likely due to small sample size ($n = 10$ after omission of missing data). The trends above are offered as signals of which tasks may be easy for authors to perform when first using Torii, and as preliminary indicators of relative task difficulty that merit further investigation.

Participants’ tool preference aligned with trends in task times. Authors felt they would be more effective using a tool like Torii for tasks like subtask *a* (9 of 10) and subtask *c* (8 of 10). Fewer believed they would be more effective using the tool for tasks like subtask *b* (5 of 10). This suggests the value of further design

iterations to improve the localized edits feature.

Authoring task. All participants (11 of 11) created tutorials with Torii within 15–30 minutes. Tutorials contained a median of six snippets ($\sigma = 1.6$) and three outputs ($\sigma = 1.3$). 10 of 11 produced the outputs authors expected; only 1 contained exceptions, which the author noticed but did not care to fix.

Usage of Torii's authoring affordances

Authors created tutorials leveraging Torii's affordances for flexible code organization. Several tutorials contained snippets that would be syntactically incomplete within a conventional notebook, but could be included without issue in Torii (3 of 11: P5, P9, P12). In all cases, incomplete snippets were class or method declarations without their bodies. Authors presented the declarations in isolation, later adding snippets with method or class bodies before generating any outputs.

A majority of authors leveraged Torii's ability to include the same code in multiple snippets. Using this feature, authors scaffolded the presentation of a class declaration, showing it multiple times, each time adding new properties or methods (6 of 11: P1, P2, P5, P9, P11, P12). A handful of authors implemented an even more intricate version of scaffolding, interleaving code that built up the class declaration with driver code that constructed and tested progressively more complex instances of the class (4 of 11: P1, P2, P9, P11).

One author presented code in reverse order from how the interpreter would need to execute it, showing a usage of a class before its declaration (P11). A sample of tutorials demonstrating these usage patterns is shown in Figure 6.5.

Desired affordances for future tools

Participants reported which of Torii's features were useful for the authoring task on a three-point scale: "very useful," "somewhat useful," "not useful," or "not applicable" (Figure 6.6).

Linked edits between the source program and snippets were described as "delightful" (P8). All but one participant found linked edits at least somewhat useful. During the maintenance tasks, all authors (10 of 10) strongly agreed that they found it easy to plan out and make linked edits.

All participants (11 of 11) found the generation of embedded outputs to be very useful, and nearly all (9 of 11) found the companion feature of live updates

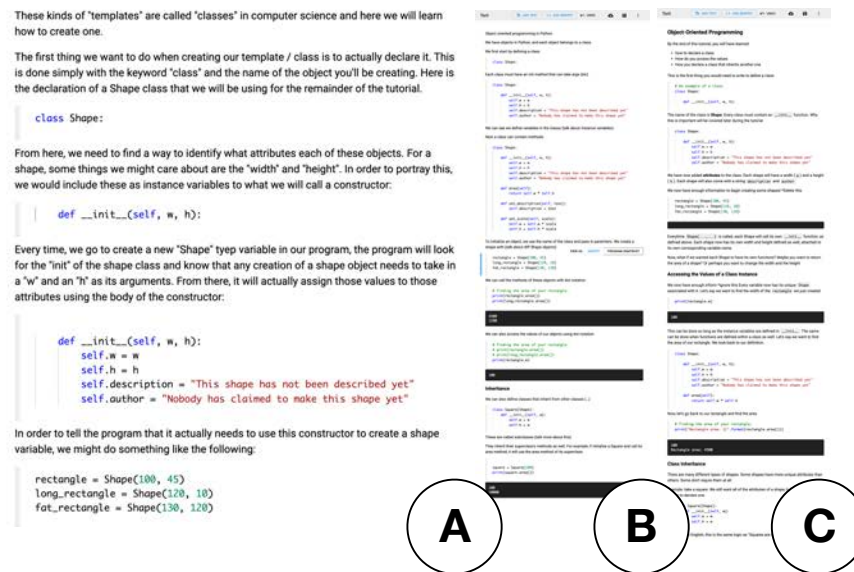


Figure 6.5: Authors created tutorials using Torii's affordances for flexible code organization. Readers are encouraged to zoom in on the tutorials above, each of which was produced by a different study participant. These tutorials show how authors included syntactically invalid snippets (A, excerpt); scaffolded the declaration of a class while repeating code across snippets (B); and interleaved code for declaring and testing a class (C).

to outputs very useful. According to one author, live updates provided them with confidence that the code above the output was correct (P7).

Snapshots and localized edits were the least useful features. One reason they were not useful is that some participants felt they did not entirely understand how snapshots—that is, the ordered assemblies of snippets used to generate outputs—were created, even after successfully authoring a tutorial with Torii (P2). Localized edits were used only once for their intended purpose of evolving code shown in earlier snippets, perhaps due to the simplicity of the tutorial authoring task or length of the study. That said, many authors (6 of 11) appropriated localized edits to disable `print` statements from previous snippets to make outputs cleaner. Some of these authors wanted a more lightweight version of localized edits that would let them add `print` statements for just one snippet, and automatically remove them from later snippets.

Authors envisioned several ways that future tools could improve the authoring experience. Tools could help participants overlay prose explanations on top of a selection of code in a snippet (P4, also requested by A1 in the interview study). Authors wanted stronger visual scent to indicate when snippets were unsynced from the source program (P3, P8). One author wondered if tools like Torii could

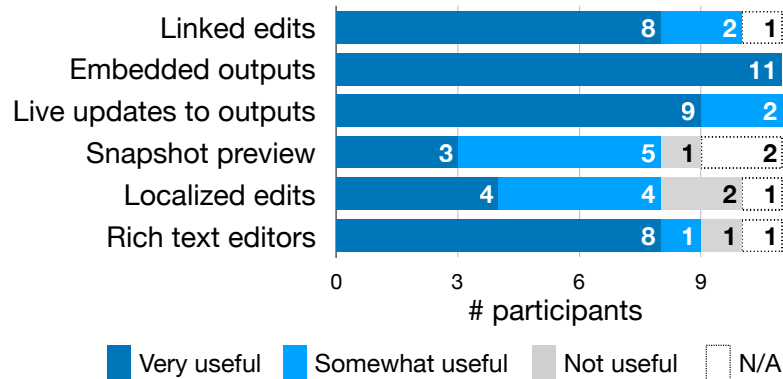


Figure 6.6: Authors found Torii’s affordances for linked editing, generating embedded outputs, and updating outputs very useful when creating tutorials. Snapshots and localized edits were less useful, and may require additional design effort in order to provide value to authors.

help them propagate edits from code to the prose explaining it (P7). Another author wished to embed visualizations of an object from the program’s state at a specific step of program execution (P4).

Conclusions

From our formative interviews, we found that authors face a unique authoring challenge of keeping source programs, snippets, and outputs consistent as they write tutorials. Our content analysis of tutorials showed that a majority of tutorials contain repeated code and generated outputs, which the tutorial’s author would need to keep consistent as they write and maintain the tutorial. Many tutorials also contained code fragments and rewritten code, indicating that tools for authoring tutorials should provide authors with considerable flexibility in how they organize a tutorial’s code as snippets.

Our in-lab usability study showed that authors can readily adopt tools like Torii to write simple tutorials with a flexibility not present in other tools. Linked edits, output generation, and live updates to outputs were valued features for the authoring task. Authors preferred Torii to a comparison tool for tasks such as making linked edits to code. Other features, like making localized edits, could benefit from further design iteration to better support authors’ use cases and mental models.

Limitations and extensions

The external validity of the formative studies is limited by our sample choice. The interviewed authors had considerable experience and wrote tutorials of ambitious scope. The content analysis focused on tutorials that were widely-referenced. It is not clear the extent to which the authoring challenges observed generalize to all authors, and all tutorials. Further research with a broader sample of authors and tutorials may surface additional authoring challenges that this chapter has overlooked.

One limitation of the in-lab usability study, common to lab studies, is that authors were not allowed to use Torii to write their own tutorials, with their own source material. We sought to mitigate this risk by asking participants to edit and reproduce real existing tutorials. Still, a holistic understanding of the tool's usability will depend on studies with longer tasks, and source programs of myriad types and languages.

Designing better tools for tutorial authoring

In the formative and in-lab studies, authors recommended affordances they would like to see in future tutorial authoring tools. These include anchoring prose explanations to selections in code snippets, linking prose to code, and allowing readers to edit and execute snippets within the tutorial.

One challenge problem for tools with Torii's execution model is providing intuitive functionality for making localized edits. We see two promising directions for future designs. First, authors may find it easier to select snippets from versioned source programs, rather than versioning individual snippets. Second, Torii's current implementation could be improved by making affordances for syncing edits more visible, and providing suitable defaults for how snippets and source programs are initially synced.

Interaction design beyond programming tutorials

Ideas from Torii's design may transfer to adjacent domains.

Torii-like tools could help software developers link code to documentation in new ways. One participant in the lab study wanted tools like Torii in their continuous integration pipeline to check that their examples in their project's documentation still functioned after the code or external dependencies changed (*P4*). By leveraging novel techniques for mining and generating documentation (e.g., Srid-

[hara et al. 2010](#); [Subramanian et al. 2014](#)), tools like Torii may also be able to support linked editing of code and prose.

Authors of tutorials in other domains might benefit from tools like Torii. One feature that could be particularly useful is Torii’s automatic updates to a tutorial’s visuals. Authors of tutorials about image manipulation, 3D modeling, and operating system configuration all create tutorials as user interface instructions interleaved with “outputs” (e.g., images, models, screenshots). Future tools could update such outputs automatically as authors edit instructions by selectively re-playing interaction logs aligned to tutorial instructions.

Chapter 7. Conclusions

“A good story cannot be devised; it has to be distilled.”

Raymond Chandler

This chapter begins by reviewing the lessons learned in the previous chapters in the context of the thesis and three claims about program distillation. Then, the dissertation concludes with a discussion of the future of program distillation tools. Opportunities are identified for future research in human-computer interaction, software engineering, and program analysis to reinvent the experience of producing and reading sample programs.

Summary of findings

My thesis is that authors can transform existing programs into sample programs more efficiently and flexibly when aided by interactive tools for selecting, simplifying, supplementing, and sequencing code. Chapters 4–6 support this thesis. They provide evidence supporting three claims in particular:

- **C1.** Interactive tools can provide functionality to help authors select, simplify, supplement, and sequence code.
- **C2.** The tools can be implemented with proven program analysis techniques.
- **C3.** The tools support effective and flexible program distillation.

The evidence supporting each of these claims is reviewed in the subsections below.

Claim I. Interactive tools can provide functionality to help authors select, simplify, supplement, and sequence code.

The tools in this dissertation help authors select, simplify, supplement, and sequence code in new ways. Chapter 3 introduced a design space of distillation tools. Let us examine the capabilities of each tool, considering what regions of the design space have now been explored (Figure 3.11). The results of usability studies for each tool will be examined in the subsection for Claim 3.

CodeScoop: *Snippet authoring via mixed-initiative selection and simplification.* CodeScoop helps authors make executable, minimal snippets up to a few dozen lines in length. The author’s code editor is augmented with an intelligent agent that helps the author *select* and *simplify* code. The agent takes initiative to repair code automatically whenever it can. It also initiates subdialogues with the author in the face of ambiguity, i.e., when it’s not clear what code to include or the best way to simplify a statement. Selection takes place through interactive expansion of a program slice. The program is simplified by replacing variables with values and test stubs mined from the program execution.

Code gathering tools: *Notebook cleaning via direct selection of code and outputs.* Code gathering tools help authors extract snippets and sequences of cells from messy computational notebooks. To *select* code, an author picks variables or outputs of interest. The tools then slice an invisible interpreter history in order to find the ordered, reduced, complete subsets of cells that produced those variables or outputs. The returned cells are *sequenced* in the order in which they were originally run. Code gathering tools *supplement* the distilled programs in two ways. First, if an author selected outputs, the tools generate a notebook which includes those outputs next to the cells that produced them. Second, the tools can generate version histories that show how the gathered cells evolved over time.

Torii: *Tutorial authoring via flexible sequencing of snippets.* Torii helps authors transform existing programs into tutorials that comprise many code snippets and outputs. Torii consists of a dedicated WYSIWYG tutorial editor, linked to a conventional code editor that contains a source program. The tool helps authors *supplement* their tutorials with outputs while permitting them to *sequence* the snippets flexibly. The tool takes initiative in keeping the source program, snippets, and outputs consistent in two ways. First, it propagates edits between source programs and snippets. Second, it updates outputs live by inferring how the snippets are supposed to be combined and executing them as they change. The most unique aspect of Torii as a distillation tool is that it implicitly creates a new version of the sample program with each snippet that the author adds to a tutorial.

Claim II. The tools are implemented with proven program analysis techniques.

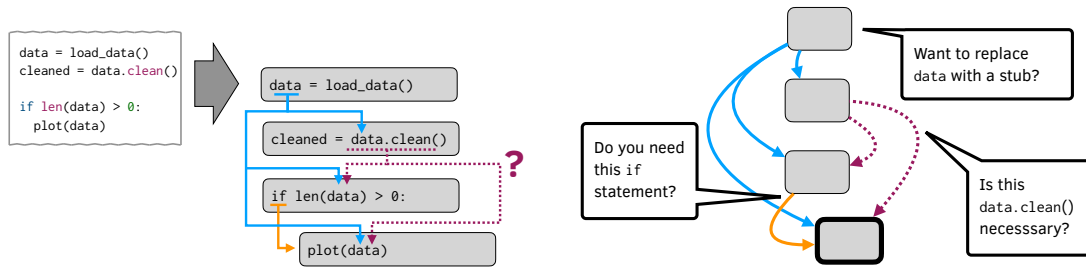
The tools are indeed implemented with proven program analysis techniques. The novel interactions for these tools are supported by extensions to the well-known techniques of program slicing (Tip et al. 1999) and linked editing (Toomim et al. 2004). The techniques are extended in four ways (Figure 7.1):

Extension 1. Incremental slicing. Slicers help programmers extract code from a program by searching for transitive dependencies of a selected line. A typical slicer may not be sure whether one line depends on another. It may also find dependencies from a line to other lines that are not strictly necessary in an extracted snippet (Figure 7.1a). In CodeScoop, a slicer asks an author for input when, in one of these situations, it's not clear whether a line is needed or not. It asks authors for input at a time when it believes the author has recently looked at the code containing the ambiguous dependencies (Figure 7.1b).

Extension 2. Pruning the dependence graph. CodeScoop helps authors reduce complexity in a snippet by replacing variables with values, and complex objects with stubs (Figure 7.1b). When a user chooses to replace a variable with a value, the tool marks the variable's node in the dependence graph with a flag, indicating to the slicer that it no longer needs to resolve dependencies for that variable.

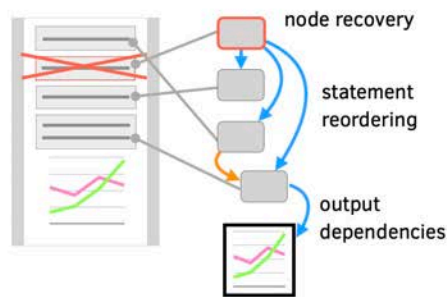
Extension 3. Slicing an interpreter history. Notebooks, at least as they appear to users, cannot be sliced. Many of the cells necessary to trace the origins of an output might have been deleted, or they might be listed out of order. Code gathering tools fix this issue, letting authors view dependencies and slice notebooks by analyzing the history of cells submitted to the interpreter. This lets the slicer recover deleted cells and put cells back in the order they were run. Outputs can be selected as a slicing criterion if dependencies are added between the output and the lines in the cell that produced it (Figure 7.1c).

Extension 4. Linking outputs to snippets through program snapshots. The linked editing paradigm in typical notebooks does not support Torii's model of tutorial authoring. Live notebooks usually require code in a cell to be complete and to be written in a way that dependencies between cells can be inferred from the code. Torii relaxes this constraint. It does this by adding a dependency from an output not to a cell that produced it, but to a snapshot of a program, composed of all snippets that appear before the output in the tutorial stacked in the order they appeared in a source program (Figure 7.1d).

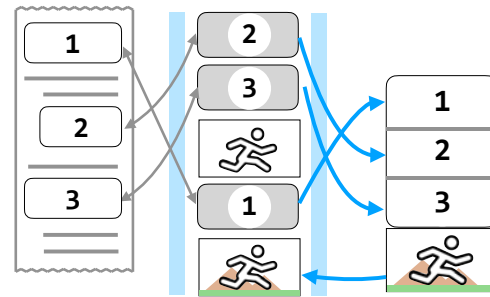


(a) A dependence graph for a short program. It contains **unambiguous dependencies**, **ambiguous ones**, and some **control dependencies that may not be strictly necessary**.

(b) CodeScoop helps authors select and simplify code by asking whether to include ambiguous dependencies at opportune times, and replacing parts of the program with stubs.



(c) Code gathering tools help authors select **ordered, complete slices** for outputs and statements in notebooks. To do so, they recover and reorder statements using the interpreter's logs.



(d) Torii links edits made to source programs, snippets, and outputs. Outputs **depend** on ordered subsets of snippets. Edits **propagate** between source programs and snippets.

Figure 7.1: A review of how program dependencies and linked edits support program distillation.

Claim III. The tools support effective and flexible program distillation.

Our first-use studies with each of the tools revealed that authors could easily adopt them. The tools also helped authors distill programs more efficiently and flexibly. The results from each study can be summarized as follows:

Snippet distillation. In a controlled experiment, programmers finished extracting examples more quickly with CodeScoop than with 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. Participants often responded to the options CodeScoop provided in different ways, suggesting that different contexts and authors prescribe different solutions.

Notebook distillation. In a qualitative usability study, participants responded positively to the code gathering tools, which automatically produce the minimal code necessary to replicate a chosen set of analysis results. During exploratory data analysis, participants primarily used code gathering as a “finishing move” to share work, but also found unanticipated uses like generating reference material, creating lightweight branches in their code, and creating summaries for multiple audiences.

Tutorial distillation. In a usability study, tutorial authors used Torii to write simple tutorials with a flexibility they would not have in other live computational notebooks. For instance, authors split code into snippets that were syntactically incomplete, displayed the same code in multiple snippets, and presented code in the reverse order it needed to run. Authors completed two of three tutorial maintenance tasks more quickly with Torii than with a comparison tool; however, these differences were not statistically significant. Authors preferred Torii to a comparison tool for tasks like making linked edits to code.

Remaining challenges and future directions

As evidenced by the white space in the design space figure (Figure 3.11), there remains much to be explored in the design of distillation tools. This section gives structure to that open space. Four research agendas are proposed to advance the creation and understanding of sample programs. These research agendas are mixed-initiative program synthesis, explorable explanations, natural language generation, and the distillation of scientific discourse.

Future directions for the individual tools introduced in this dissertation can be found on pages 81 (Chapter 4), 100 (Chapter 5), and 124 (Chapter 6).

Mixed-initiative program synthesis

The tools in this dissertation help authors simplify code by letting them replace primitive variables with values and object variables with stubs. Could distillation tools help authors with more ambitious simplifications, like replacing entire sets of statements at once with readable, idiomatic equivalent statements?

Simplification of a sample program could be posed as a task of mixed-initiative synthesis by sketching (Solar-Lezama 2008). To simplify a program, an author would first mark up sets of statements in a sample program they want the synthesizer to rewrite. This would generate a program sketch, consisting of the sample program minus the selected statements. A synthesizer could be given that sketch,

along with the sample program as a reference implementation, as a specification for synthesis. Such a synthesizer might be built using recent techniques for synthesizing imperative code (e.g., Galenson et al. 2014; Feng et al. 2017). The question is, how could such synthesizers be tailored to support real-time synthesis and generation of program alternatives in the context of distillation? And how can authors express constraints for how they wish the rewritten code to appear?

Compatible with the goals of program distillation, program synthesizers have been built to generate programs that are both readable and modifiable by end users (Little et al. 2007; Mayer et al. 2015; Chasins et al. 2018; Drosos et al. 2020). These synthesizers are typically one-shot: the synthesizer produces a program, and any further modifications are performed by the programmer alone. The vision here is that instead of one-shot synthesis, a programmer produces a program through iterative, mixed-initiative, piecemeal synthesis until they have refined the program into just the one they wish to read and share with others.

Authoring tools for explorable tutorials

The authors we interviewed in Chapter 6 wanted to let readers tinker with the code in the snippets of their tutorials and observe how their changes impacted the program’s behavior. To borrow a term from Victor (2011) and his essay on new media for active reading, authors wanted their tutorials to be *explorable*.

What would it mean to design effective explorable programming tutorials? The design space of explorable programming tutorials is broad. A tutorial could let readers directly edit code and view its output. It could let readers tinker with the output if the output is interactive, like a user interface or visualization. The tutorial could even provide interactive visualizations of program execution like PythonTutor (Guo 2013). Which of these and other capabilities do authors want their tutorials to have, and which ones would actually help readers? Would readers notice them? Would they use them? Would interactivity help readers learn from, design with, or reuse the sample program? While such questions have been explored recently for interactive articles on the web (Conlen et al. 2019), little is known about how to make explorable programming tutorials effective.

What might a toolkit for authoring explorable programming tutorials look like? One key technical obstacle that tools could help with is figuring out how to execute code in a tutorial as a reader tinkers with it. This is a challenging problem for three reasons. First, the order in which code is displayed in a tutorial does not always match the order in which it needs to be run (see Chapter 6). Second, a program may take more time to compile or execute than a reader wishes to wait. Third, a program may be written in a language that cannot be executed in the browser. To

create explorable tutorials, authors will need systems that support the execution of code despite these limitations. One solution to all of these problems is a tool that discovers interesting changes that a reader might want to make to a program automatically, before the tutorial is deployed. The tool could save the program's behavior after making each of these changes, serialize the behavior, and upload it along with the tutorial. This would allow readers to explore an interesting, if constrained, set of program variants with immediate feedback.

Natural language generation

Usable tools have been developed to support linked editing of code clones, and code and outputs (Chapter 3, page 47). Could distillation tools help authors explain sample programs by supporting linked editing of code and text?

This would require advances in algorithm design and interaction design. Algorithms like Baker (Subramanian et al. 2014) and FreshDoc (Lee et al. 2019) already reliably detect links between code elements like API methods and classes and documentation that refers to them. Techniques could be developed to detect links between sentences in programming documentation and the program statements or methods they refer to. Assuming accurate links have been found between code and texts, interactions for linked editing might take multiple forms. For instance, linters could continuously scan explanations of a sample program and highlight text that has gone out of sync with the code. Changes to a variable name in the code could trigger updates to references to that variable name in the text. Each of these ideas requires iterative design to assess whether they work for authors, especially if links between code and text are sometimes inaccurate.

Recently, techniques have been developed for generating natural language explanations of programs (e.g., Sridhara et al. 2010; Iyer et al. 2016) (see also Figure 7.2). In the spirit of distillation, how could these techniques be extended to let authors shape the generated explanations? Could the techniques allow authors to provide input to fine-tune the vocabulary used in explanations and express the level of detail that should be included in explanations?

The distillation of scientific discourse and beyond

Could tools one day aid the distillation of other types of documents, like research manuscripts? For instance, perhaps this dissertation could be distilled into two blog posts for two distinct audiences. One blog post could describe the role of program slicing in distillation tools. Another one could summarize empirical findings about how authors create programming tutorials.

```
def get_video_data(video_page_url):
    video_data = {}
    response = requests.get(root_url + video_page_url)
    soup = bs4.BeautifulSoup(response.text)
    video_data['title'] = soup.select('div#videobox h3')[0].get_text()
    video_data['speakers'] = [a.get_text() for a in soup.select('div#sidebar a[href^="/speaker']')]
    video_data['youtube_url'] = soup.select('div#videobox a[href^="https://www.youtube.com/watch?v="])[0].get_text()
```

A few things to note about this function

- The URLs returned from the scraping are the same as the original page.
- The session title is obtained from the `<h3>` because the `select()` call returns a list.
- The speaker names and YouTube links are obtained from the `select()` call.

Now all that remains is to scrape the video URL. This is very simple to write as a continuation of the function. We can also scrape the likes and dislikes.

You found a CSS selector.

The selector `'div#sidebar a[href^="/speaker']` chooses links with URLs starting with `/speaker` from a container with the ID `'sidebar'`.

If you haven't seen them before, selectors pick sections of HTML pages by their names or properties. Once you've 'grabbed' elements with a selector, you can manipulate them, like changing their appearance or text.

Here's an example of what this selector will find:

```
<div id="sidebar">
  <a href="/speaker">
  </a>
</div>
```

```
def get_video_data(video_page_url):
```

Figure 7.2: How can tools help authors generate context-relevant explanations of sample programs? Pictured is an explanation of a CSS selector in a tutorial, generated by the *Tutorons* (Head et al. 2015) system. It includes a usage sample and a prose explanation of the function of the code. Could a system like *Tutorons* allow authors to influence the vocabulary or level of detail of generated explanations?

To distill in a new domain, interactions for selecting, simplifying, supplementing, and sequencing code must be ported and refined. Let us consider the task of distilling a paper containing mathematics. Like programs, such papers define symbols and use these symbols to define other symbols. A distillation tool could help an author extract a mathematical line of reasoning from a paper. The author would select an equation of interest, and the tool would slice the paper on the variable used. If it's ambiguous whether an equation really requires a symbol, the distillation tool can ask the author for input. Equations could be simplified by renaming symbols to canonical names. The distilled paper could be supplemented with prose and figures that refer to the symbols. The result would be a complete, if stilted, overview of the sections of a paper relevant to an equation of interest.

Scientific discourse is singled out as a potential domain for distillation because scientists already distill their knowledge into blog posts, presentations, and grants, and because readers read papers strategically and piecemeal when searching for information of interest (Bazerman 1985). Research continues to reveal new ways of extracting semantic understanding (e.g., Beltagy et al. 2019) and structured knowledge (e.g., Siegel et al. 2018) from research papers. That said, perhaps tools could help authors distill yet other documents that lack formal representations, like the book *Moby Dick* or a massive dataset like the U.S. census data. Design opportunities for distillation tools can be revealed by studying authors as they distill their own works into derivative forms for new audiences.

Closing remarks: Humans, compilers, and creativity

The vision of this dissertation is that programmers can more effectively create sample programs if they do so in dialogue with their tools. For each of the tools introduced, authors indicate patterns they want to share in their code, and their tools help them select code of interest, and simplify, supplement, and sequence it so that it can be most readable and reusable.

The tools were designed to amplify the human voice in the process of transforming programs meant to be read by human audiences. One is reminded of the words of Tony Hoare (1973) in his essay *Hints on Programming Language Design*:

“It is of course possible for a compiler or service program to expand the abbreviations, fill in the defaults, and make explicit the assumptions. But in practice, experience shows that it is very unlikely that the output of a computer will ever be more readable than its input, except in such trivial but important aspects as improved indentation.”

The role of distillation tools is to capture human input so that sample programs can be infused with meaning that cannot otherwise be derived from the source program itself. This dissertation shows that when given an opportunity to exercise their voice, authors make use of it. In the future, tools like these will help authors disseminate their knowledge more efficiently and flexibly, in the domains of programming and beyond.

Bibliography

Agrawal, Hiralal and Joseph R. Horgan. “Dynamic Program Slicing.” *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, 1990, pp. 246–256 (cited on page 44).

Allamanis, Miltiadis and Charles Sutton. “Mining Idioms from Source Code.” *Proceedings of the International Symposium on Foundations of Software Engineering*. 2014, pp. 472–483 (cited on page 24).

Allen, James F. “Mixed-initiative interaction.” *IEEE Intelligent Systems and their Applications* 14.5 (1999), pp. 14–16 (cited on page 52).

Anderson, John R., Robert Farrell, and Ron Sauers. “Learning to program in LISP.” *Cognitive Science* 8.2 (1984), pp. 87–129 (cited on pages 13, 14).

Anderson, Paul and Tim Teitelbaum. “Software Inspection using CodeSurfer.” *Workshop on Inspection in Software Engineering*. 2001, pp. 4–11 (cited on page 44).

Backus, J. W., R. J. Beeber, S. Best, R. Goldberg, H. L. Herrick, R. A. Hughes, L. B. Mitchell, R. A. Nelson, R. Nutt, D. Sayre, P. B. Sheridan, H. Stern, and I. Ziller. *Fortran: Automatic Coding System for the IBM 704 EDPM*. 1956 (cited on page 22).

Baltes, Sebastian and Stephan Diehl. “Usage and attribution of Stack Overflow code snippets in GitHub projects.” *Empirical Software Engineering* 24 (2019), pp. 1259–1295 (cited on page 12).

Barik, Titus, Yoonki Song, Brittany Johnson, and Emerson Murphy-Hill. “From Quick Fixes to Slow Fixes: Reimagining Static Analysis Resolutions to Enable Design Space Exploration.” *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2016, pp. 211–221 (cited on page 46).

Bazerman, Charles. “Physicists Reading Physics: Schema-Laden Purposes and Purpose-Laden Schema.” *Written Communication* 2.1 (1985), pp. 3–23 (cited on page 133).

Beck, Kent. “Aim, fire.” *IEEE Software* 18.5 (2001), pp. 87–89 (cited on page 12).

Beltagy, Iz, Kyle Lo, and Arman Cohan. “SciBERT: A Pretrained Language Model for Scientific Text.” *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2019, pp. 3615–3620 (cited on page 133).

Bentley, Jon, Don Knuth, and Doug McIlroy. “Programming pearls: A literate program.” *Communications of the ACM* 29.6 (1986), pp. 471–483 (cited on page 33).

Binkley, Dave, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. “The impact of identifier style on effort and comprehension.” *Empirical Software Engineering* 18 (2013), pp. 219–276 (cited on page 11).

Binkley, David and Mark Harman. “A Survey of Empirical Results on Program Slicing.” *Advances in Computers* 62 (2004). Ed. by Marvin V. Zelkowitz, pp. 105–178 (cited on page 44).

Boehm-Davis, Deborah A., Robert W. Holt, and Alan C. Schultz. “The role of program structure in software maintenance.” *International Journal of Man-Machine Studies* 36.1 (1992), pp. 21–63 (cited on page 10).

Booth, Andrew D. and Kathleen H. V. Booth. *Automatic digital calculators*. 3rd ed. Butterworths, 1956 (cited on page 22).

Boshernitsan, Marat, Susan L. Graham, and Marti A. Hearst. “Aligning Development Tools with the Way Programmers Think About Code Changes.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2007, pp. 567–576 (cited on page 46).

Bragdon, Andrew, 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.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 2503–2512 (cited on page 82).

Brandt, Joel, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. “Example-Centric Programming: Integrating Web Search into the Development Environment.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 513–522 (cited on page 58).

Brandt, Joel, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. “Two Studies of Opportunistic Programming: Interleaving Web Foraging,

Learning, and Writing Code.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1589–1598 (cited on pages 12, 13).

Brandt, Joel, Vignan Pattamatta, William Choi, Ben Hsieh, and Scott R. Klemmer. *Rehearse: Helping Programmers Adapt Examples by Visualizing Execution and Highlighting Related Code*. Tech. rep. Stanford University, 2010 (cited on page 45).

Brooks, Ruven. “Towards a theory of the comprehension of computer programs.” *International Journal of Man-Machine Studies* 18.6 (1983), pp. 543–554 (cited on page 8).

Burden, Håkan and Rogardt Heldal. “Natural Language Generation from Class Diagrams.” *Proceedings of the International Workshop on Model-Driven Engineering, Verification and Validation*. Article 8. ACM, 2011 (cited on page 50).

Burg, Brian, Richard Bailey, Amy J. Ko, and Michael D. Ernst. “Interactive Record/Replay for Web Application Debugging.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2013, pp. 473–483 (cited on page 45).

Burg, Brian, Amy J. Ko, and Michael D. Ernst. “Explaining Visual Changes in Web Interfaces.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2015, pp. 259–269 (cited on page 45).

Burkhardt, Jean-Marie and Françoise Detienne. “An empirical study of software reuse by experts in object-oriented design.” *Proceedings of the International Conference on Human-Computer Interaction*. Springer, 1995, pp. 133–138 (cited on page 12).

Buse, Raymond P.L. and Westley Weimer. “Synthesizing API Usage Examples.” *Proceedings of the International Conference on Software Engineering*. IEEE, 2012, pp. 782–792 (cited on pages 14, 15, 24–26, 111).

Buse, Raymond P.L. and Westley R. Weimer. “Learning a Metric for Code Readability.” *IEEE Transactions on Software Engineering* 36.4 (2010), pp. 546–558 (cited on pages 25, 26).

Busjahn, Teresa, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. “Eye Movements in Code Reading: Relaxing the Linear Order.” *Proceedings of the International Conference on Program Comprehension*. IEEE, 2015, pp. 255–265 (cited on page 8).

Card, Stuart K., Jock D. Mackinlay, and George G. Robertson. “A Morphological Analysis of the Design Space of Input Devices.” *Transactions on Information Systems* 9.2 (1991), pp. 99–122 (cited on page 51).

Carroll, John M. *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*. The MIT Press, 1990 (cited on page 16).

Chasins, Sarah E., Maria Mueller, and Rastislav Bodik. “Rousillon: Scraping Distributed Hierarchical Web Data.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2018, pp. 963–975 (cited on pages 28, 131).

Chen, Charles H. and Philip J. Guo. “Improv: Teaching Programming at Scale via Live Coding.” *Proceedings of the Conference on Learning at Scale*. Article 9. ACM, 2019 (cited on pages 40, 41).

Chen, Zhifei, Lin Chen, Yuming Zhou, Zhaogui Xu, William C. Chu, and Baowen Xu. “Dynamic Slicing of Python Programs.” *Proceedings of the International Computers, Software and Applications Conference*. IEEE, 2014, pp. 219–228 (cited on page 73).

Chi, Pei-Yu (Peggy), Sen-Po Hu, and Yang Li. “Doppio: Tracking UI Flows and Code Changes for App Development.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Paper 455. ACM, 2018 (cited on pages 29, 30).

Childs, Bart. “Literate Programming, A Practioner’s View.” *TUGboat* 13.3 (1992) (cited on page 32).

Childs, Bart, Deborah Dunn, and William Lively. “Teaching CS/1 Courses in a Literate Manner.” *TUGboat* 16.3 (1995), p. 8 (cited on page 34).

Conlen, Matthew, Alex Kale, and Jeffrey Heer. “Capture & Analysis of Active Reading Behaviors for Interactive Articles on the Web.” *Proceedings of the Eurographics Conference on Visualization*. John Wiley & Sons, Ltd., 2019, pp. 687–698 (cited on page 131).

Cooper, Alan, Robert Reimann, and David Cronin. *About Face 3: The Essentials of Interaction Design*. Wiley Publishing, Inc., 2007 (cited on page 51).

Cuoq, Pascal, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. “Frama-C: A Software Analysis Perspective.” *Proceedings of the International Conference on Software Engineering and Formal Methods*. Springer, 2012, pp. 233–247 (cited on page 44).

Cypher, Allen, ed. *Watch What I Do: Programming by Demonstration*. The MIT Press, 1993 (cited on page 28).

Dagenais, Barthélémy and Martin P. Robillard. “Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors.” *Proceedings of the International Symposium on Foundations of Software Engineering*. ACM, 2010, pp. 127–136 (cited on pages 17, 103).

Dagenais, Barthélémy and Martin P. Robillard. “Using Traceability Links to Recommend Adaptive Changes for Documentation Evolution.” *IEEE Transactions on Software Engineering* 40.11 (2014), pp. 1126–1146 (cited on page 49).

Daka, Ermira, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. “Modeling Readability to Improve Unit Tests.” *Proceedings of the Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 107–118 (cited on page 46).

DeLine, Robert and Danyel Fisher. “Supporting Exploratory Data Analysis with Live Programming.” *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2015, pp. 111–119 (cited on pages 37, 38, 44).

DeLine, Robert, Danyel Fisher, Badrish Chandramouli, Jonathan Goldstein, Michael Barnett, James Terwilliger, and John Wernsing. “Tempe: Live Scripting for Live Data.” *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2015, pp. 137–141 (cited on pages 37, 39, 49).

DeNero, John. *Composing Programs*. <https://composingprograms.com/>. Last accessed May 5, 2020 (cited on page 50).

Dorn, Brian and Mark Guzdial. “Learning on the Job: Characterizing the Programming Knowledge and Learning Strategies of Web Designers.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 703–712 (cited on page 13).

Drosos, Ian, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. “Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Paper 315. ACM, 2020 (cited on pages 28, 131).

Dunsmore, Alastair, Marc Roper, and Murray Wood. “Object-Oriented Inspection in the Face of Delocalisation.” *Proceedings of the International Conference on Software Engineering*. ACM, 2000, pp. 467–476 (cited on page 10).

Eclipse Foundation. *Quick Fix and Quick Assist*. https://help.eclipse.org/2020-03/topic/org.eclipse.jdt.doc.user/concepts/concept-quickfix-assist.htm?cp=1_2_5. Last accessed May 6, 2020 (cited on page 46).

Fast, Ethan and Michael S. Bernstein. “Meta: Enabling Programming Languages to Learn from the Crowd.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2016, pp. 259–270 (cited on pages 28, 30).

Fast, Ethan, Daniel Steffee, Lucy Wang, Joel Brandt, and Michael S. Bernstein. “Emergent, Crowd-scale Programming Practice in the IDE.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2014, pp. 2491–2500 (cited on page 24).

Feigenspan, Janet, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachsel, Maria Papendieck, Thomas Leich, and Gunter Saake. “Do background colors improve program comprehension in the #ifdef hell?” *Empirical Software Engineering* 18 (2013), pp. 699–745 (cited on page 11).

Felleisen, Matthias, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: an Introduction to Programming and Computing*. 2nd ed. The MIT Press, 2018 (cited on page 12).

Feng, Yu, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. “Component-Based Synthesis for Complex APIs.” *Proceedings of SIGPLAN Symposium on Principles of Programming Languages*. ACM, 2017, pp. 599–612 (cited on page 131).

Fitzmaurice, George W., Hiroshi Ishii, and William Buxton. “Bricks: Laying the Foundations for Graspable User Interfaces.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 1995, pp. 442–449 (cited on page 51).

Ford, Denae, Justin Smith, Philip J. Guo, and Chris Parnin. “Paradise Unplugged: Identifying Barriers for Female Participation on Stack Overflow.” *Proceedings of the International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 846–857 (cited on page 17).

Fourney, Adam and Meredith Ringel Morris. “Enhancing Technical Q&A Forums with CiteHistory.” *Proceedings of the International Conference on Weblogs and Social Media*. The AAAI Press, 2013 (cited on page 49).

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. 2nd ed. Addison-Wesley Professional, 2018 (cited on page 46).

Fraser, Gordon and Andreas Zeller. “Exploiting Common Object Usage in Test Case Generation.” *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 80–89 (cited on page 46).

Galenson, Joel, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. “CodeHint: Dynamic and Interactive Synthesis of Code Snippets.” *Proceedings of the International Conference on Software Engineering*. ACM, 2014, pp. 653–663 (cited on pages 27, 30, 131).

Ginosar, Shiry, Luis Fernando De Pombo, Maneesh Agrawala, and Björn Hartmann. “Authoring Multi-Stage Code Examples with Editable Code Histories.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2013, pp. 485–494 (cited on pages 17, 18, 40–42, 58).

Glassman, Elena L., Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. “OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale.” *ACM Transactions on Computer-Human Interaction* 22.2 (2015), 7:1–7:35 (cited on pages 24–26).

Glassman, Elena L., Tianyi Zhang, Björn Hartmann, and Miryung Kim. “Visualizing API Usage Examples at Scale.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Paper 580. ACM, 2018 (cited on page 47).

Goffi, Alberto, Alessandra Gorla, Andrea Mattavelli, Mauro Pezzè, and Paolo Tonella. “Search-Based Synthesis of Equivalent Method Sequences.” *Proceedings of the International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 366–376 (cited on page 47).

Goldman, Max and Robert C. Miller. “Codetrail: Connecting source code and web resources.” *Journal of Visual Languages and Computing* 20 (2009), pp. 223–235 (cited on page 49).

Google. *Codewalk: How to write a Codewalk*. <https://golang.org/doc/codewalk/codewalk/>. Last accessed May 6, 2020 (cited on page 38).

Google. *Codewalk: Share Memory By Communicating*. <https://golang.org/doc/codewalk/sharemem/>. Last accessed May 6, 2020 (cited on page 39).

Gordon, Mitchell and Philip J. Guo. “Codepourri: Creating Visual Coding Tutorials Using a Volunteer Crowd of Learners.” *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2015, pp. 13–21 (cited on pages 38, 39).

Gross, Paul A., Micah S. Herstand, Jordana W. Hodges, and Caitlin L. Kelleher. “A Code Reuse Interface for Non-Programmer Middle School Students.” *Proceedings of the International Conference on Intelligent User Interfaces*. ACM, 2010, pp. 219–228 (cited on page 45).

Gulwani, Sumit, Oleksandr Polozov, and Rishabh Singh. “Program synthesis.” *Foundations and Trends in Programming Languages* 4.1-2 (2017), pp. 1–119 (cited on pages 26, 28).

Guo, Philip J. “Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education.” *Proceedings of the Technical Symposium on Computer Science Education*. ACM, 2013, pp. 579–584 (cited on pages 50, 131).

Hannebauer, Christoph, Marc Hesenius, and Volker Gruhn. “Does syntax highlighting help programming novices?” *Empirical Software Engineering* 23 (2018), pp. 2795–2828 (cited on page 11).

Harman, Mark, David Binkley, and Sebastian Danicic. “Amorphous program slicing.” *The Journal of Systems and Software* 68.1 (2003), pp. 45–64 (cited on page 46).

Hartmann, Björn, Mark Dhillon, and Matthew K. Chan. “HyperSource: Bridging the Gap Between Source and Code-Related Web Sites.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2011, pp. 2207–2210 (cited on page 49).

Hartmann, Björn, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. “Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2008, pp. 91–100 (cited on pages 48, 51).

Head, Andrew, Codanda Appachu, Marti A. Hearst, and Björn Hartmann. “Tutorons: Generating Context-Relevant, On-Demand Explanations and Demonstrations of Online Code.” *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2015, pp. 3–12 (cited on page 133).

Head, Andrew, Elena L. Glassman, Björn Hartmann, and Marti A. Hearst. “Interactive Extraction of Examples from Existing Code.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Paper 85. ACM, 2018 (cited on page 5).

Head, Andrew, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. “Managing Messes in Computational Notebooks.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Paper 270. ACM, 2019 (cited on page 5).

Head, Andrew, Jason Jiang, James Smith, Marti A. Hearst, and Björn Hartmann. “Composing Flexibly-Organized Step-by-Step Tutorials from Linked Source Code, Snippets, and Outputs.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. To appear. ACM, 2020 (cited on pages 5, 109, 113).

Hempel, Brian, Justin Lubin, and Ravi Chugh. “Sketch-n-Sketch: Output-Directed Programming for SVG.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2019, pp. 281–292 (cited on page 28).

Hempel, Brian, Justin Lubin, Grace Lu, and Ravi Chugh. “Deuce: A Lightweight User Interface for Structured Editing.” *Proceedings of the International Conference on Software Engineering*. ACM, 2018, pp. 654–664 (cited on page 46).

Heo, Kihong, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. “Effective Program Debloating via Reinforcement Learning.” *Proceedings of the SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 380–394 (cited on page 45).

Hibschman, Joshua and Haoqi Zhang. “Telescope: Fine-Tuned Discovery of Interactive Web UI Feature Implementation.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2016, pp. 233–245 (cited on page 45).

Hibschman, Joshua and Haoqi Zhang. “Unravel: Rapid Web Application Reverse Engineering via Interaction Recording, Source Tracing, and Library Detection.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2015, pp. 270–279 (cited on page 45).

Hoare, C. A. R. *Hints on Programming Language Design*. Tech. rep. Stanford University, 1973 (cited on page 134).

Hoffmann, Raphael, James Fogarty, and Daniel S. Weld. “Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2007, pp. 13–22 (cited on pages 13, 58).

Hoffswell, Jane, Arvind Satyanarayan, and Jeffrey Heer. “Augmenting Code with In Situ Visualizations to Aid Program Understanding.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Paper 532. ACM, 2018 (cited on pages 50, 51).

Holmes, Reid and Robert J. Walker. “Systematizing Pragmatic Software Reuse.” *ACM Transactions on Software Engineering and Methodology* 21.4 (2012) (cited on pages 44, 45).

Horton, Eric and Chris Parnin. “Gistable: Evaluating the Executability of Python Code Snippets on GitHub.” *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2018, pp. 217–227 (cited on page 47).

Horwitz, Susan, Thomas Reps, and David Binkley. “Interprocedural Slicing Using Dependence Graphs.” *ACM Transactions on Programming Languages and Systems* 12.1 (1990), pp. 26–60 (cited on page 44).

Hou, Daqing, Patricia Jablonski, and Ferosh Jacob. “CnP: Towards an Environment for the Proactive Management of Copy-and-Paste Programming.” *Proceedings of the International Conference on Program Comprehension*. IEEE, 2009, pp. 238–242 (cited on page 48).

Iyer, Srinivasan, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. “Summarizing Source Code using a Neural Attention Model.” *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2016, pp. 2073–2083 (cited on pages 50, 132).

Java Debug Interface. <https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/index.html>. Last accessed May 6, 2020 (cited on page 71).

Jayaraman, Ganeshan, Venkatesh Prasad Ranganath, and John Hatcliff. “Kaveri: Delivering the Indus Java Program Slicer to Eclipse.” *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*. Springer, 2005, pp. 269–272 (cited on page 44).

Jison. <http://jison.org>. Last accessed May 6, 2020 (cited on page 93).

Johnson, Maggie and Max Senges. “Learning to be a programmer in a complex organization.” *Journal of Workplace Learning* 22.3 (2010), pp. 180–194 (cited on pages 12, 38).

Jones, John Chris. *Design Methods*. 2nd ed. John Wiley & Sons, Ltd., 1992 (cited on page 51).

Jupyter. <https://jupyter.org/>. Last accessed May 6, 2020 (cited on pages 34, 37, 39, 86).

Jupyter Contrib Team. *Unofficial Jupyter Notebook Extensions*. <https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/>. Last accessed May 6, 2020 (cited on page 86).

Kamermans, Mike “Pomax”. *Let’s make a Mario game*. <http://processingjs.nihongoresources.com/test/PjsGameEngine/docs/tutorial/mario.html>. Last accessed May 6, 2020 (cited on page 1).

Kamimura, Manabu and Gail C. Murphy. “Towards Generating Human-Oriented Summaries of Unit Test Cases.” *Proceedings of the International Conference on Program Comprehension*. IEEE, 2013, pp. 215–218 (cited on page 50).

Kandel, Sean, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. “Enterprise Data Analysis and Visualization: An Interview Study.” *Transactions on Visualization and Computer Graphics* 18.12 (2012), pp. 2917–2926 (cited on pages 36, 85).

Kang, Hyeonsu and Philip J. Guo. “Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2017, pp. 737–745 (cited on page 49).

Kelley, Kyle and Brian Granger. “Jupyter Frontends: From the Classic Jupyter Notebook to JupyterLab, nteract, and Beyond.” *JupyterCon*. Video. 2017. <https://www.youtube.com/watch?v=YKmJvHjTGAM> (cited on pages 85, 86).

Kery, Mary Beth, Amber Horvath, and Brad Myers. “Variolite: Supporting Exploratory Programming by Data Scientists.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2017, pp. 1265–1276 (cited on pages 35, 86).

Kery, Mary Beth, Bonnie E. John, Patrick O’Flaherty, Amber Horvath, and Brad A. Myers. “Towards Effective Foraging by Data Scientists to Find Past Analysis Choices.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Paper 92. ACM, 2019 (cited on pages 36, 39).

Kery, Mary Beth and Brad A. Myers. “Exploring Exploratory Programming.” *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2017, pp. 25–29 (cited on pages 35, 85).

Kery, Mary Beth and Brad A. Myers. “Interactions for Untangling Messy History in a Computational Notebook.” *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2018, pp. 147–155 (cited on pages 36, 37, 39).

Kery, Mary Beth, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. “The Story in the Notebook: Exploratory Data Science using a Literate

Programming Tool.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Paper 174. ACM, 2018 (cited on pages 35, 36, 85, 87).

Khandwala, Kandarp and Philip J. Guo. “Codemotion: Expanding the Design Space of Learner Interactions with Computer Programming Tutorial Videos.” *Proceedings of the Conference on Learning at Scale*. Article 57. ACM, 2018 (cited on page 42).

Kim, Ada S. and Amy J. Ko. “A Pedagogical Analysis of Online Coding Tutorials.” *Proceedings of the Technical Symposium on Computer Science Education*. ACM, 2017, pp. 321–326 (cited on pages 16, 18, 19, 105).

Kim, Jinhan, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. “Adding Examples into Java Documents.” *Proceedings of the International Conference on Automated Software Engineering*. IEEE, 2009, pp. 540–544 (cited on page 24).

Kim, Jinhan, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. “Towards an Intelligent Code Search Engine.” *Proceedings of the AAAI Conference on Artificial Intelligence*. The AAAI Press, 2010, pp. 1358–1363 (cited on pages 23, 24, 26).

Kim, Miryung, Lawrence Bergman, Tessa Lau, and David Notkin. “An Ethnographic Study of Copy and Paste Programming Practices in OOPL.” *Proceedings of the International Symposium on Empirical Software Engineering*. IEEE, 2004, pp. 83–92 (cited on page 48).

Knitr. <https://yihui.org/knitr/>. Last accessed May 6, 2020 (cited on page 34).

Knuth, Donald E. “Literate Programming.” *The Computer Journal* 27.2 (1984), pp. 97–111 (cited on pages ix, 31–34, 103).

Knuth, Donald E. *T_EX: The Program*. Addison-Wesley, 1986 (cited on page ix).

Ko, Amy and Brad A. Myers. “Finding Causes of Program Output with the Java Whyline.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2009, pp. 1569–1578 (cited on pages 44, 45).

Koenemann, Jürgen and Scott P. Robertson. “Expert problem solving strategies for program comprehension.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 1991, pp. 125–130 (cited on page 8).

Kojouharov, Chris, Aleksey Solodovnik, and Gleb Naumovich. “JTutor: An Eclipse Plug-in Suite for Creation and Replay of Code-based Tutorials.” *Proceedings of the Eclipse Technology eXchange Workshop*. ACM, 2004, pp. 27–31 (cited on pages 40, 41, 58).

Krämer, Jan-Peter, Joel Brandt, and Jan Borchers. “Using Runtime Traces to Improve Documentation and Unit Test Authoring for Dynamic Languages.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2016, pp. 3232–3237 (cited on pages 28, 30).

Krippendorff, Klaus. *Content Analysis: An Introduction to Its Methodology*. 3rd ed. SAGE Publications, Inc., 2013 (cited on page 109).

Lange, Beth M. and Thomas G. Moher. “Some strategies of reuse in an object-oriented programming environment.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 1989, pp. 69–73 (cited on pages 13, 48).

Lau, Tessa. “Why Programming by Demonstration Systems Fail: Lessons learned for Usable AI.” *AI Magazine* 30.4 (2009), pp. 65–67 (cited on page 28).

Lawrie, Dawn, Christopher Morrell, Henry Feild, and David Binkley. “What’s in a Name? A Study of Identifiers.” *Proceedings of the International Conference on Program Comprehension*. IEEE, 2006, pp. 3–12 (cited on page 10).

Le Goues, Claire, Michael Pradel, and Abhik Roychoudhury. “Automated Program Repair.” *Communications of the ACM* 62.12 (2019), pp. 56–65 (cited on page 47).

Lee, Seonah, Rongxin Wu, Shing-Chi Cheung, and Sungwon Kang. “Automatic Detection and Update Suggestion for Outdated API Names in Documentation.” *IEEE Transactions on Software Engineering* (2019). To appear (cited on pages 49, 132).

Lee, Yun Young, Nicholas Chen, and Ralph E. Johnson. “Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation.” *Proceedings of the International Conference on Software Engineering*. IEEE, 2013, pp. 23–32 (cited on page 46).

Leshed, Gilly, Eben M. Haber, Tara Matthews, and Tessa Lau. “CoScripter: Automating & Sharing How-To Knowledge in the Enterprise.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2008, pp. 1719–1728 (cited on page 28).

Lethbridge, Timothy C., Janice Singer, and Andrew Forward. “How Software Engineers Use Documentation: The State of the Practice.” *IEEE Software* 20.6 (2003), pp. 35–39 (cited on page 48).

Letovsky, Stanley and Elliot Soloway. “Delocalized Plans and Program Comprehension.” *IEEE Software* 3.3 (1986), pp. 41–49 (cited on page 10).

Lieber, Tom, Joel Brandt, and Robert C. Miller. “Addressing Misconceptions About Code with Always-On Programming Visualizations.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2014, pp. 2481–2490 (cited on page 50).

Lin, Janet Mei-Chuen and Cheng-Chih Wu. “Suggestions for content selection and presentation in high school computer textbooks.” *Computers & Education* 48.3 (2007), pp. 508–521 (cited on page 19).

Linden, Dirk van der, Emma Williams, Joseph Hallett, and Awais Rashid. “The impact of surface features on choice of (in)secure answers by Stackoverflow readers.” *IEEE Transactions on Software Engineering* (2020). To appear (cited on page 13).

Linn, Marcia C. and Michael J. Clancy. “The case for case studies of programming problems.” *Communications of the ACM* 35.3 (1992), pp. 121–132 (cited on pages 16, 17).

Little, Greg, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. “Koala: Capture, Share, Automate, Personalize Business Processes on the Web.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. This paper describes an earlier version of the *CoScripter* tool called *Koala*. ACM, 2007, pp. 943–946 (cited on pages 30, 131).

Littman, David C., Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. “Mental Models and Software Maintenance.” *The Journal of Systems and Software* 7.4 (1987), pp. 341–355 (cited on page 8).

MacLeod, Laura, Andreas Bergen, and Margaret-Anne Storey. “Documenting and sharing software knowledge using screencasts.” *Empirical Software Engineering* 22 (2017), pp. 1478–1507 (cited on pages 17, 18).

Mahoney, Mark. “Storyteller: a Tool for Creating Worked Examples.” *Journal of Computing Sciences in Colleges* 34.1 (2018), pp. 137–144 (cited on pages 40–42).

Mamykina, Lena, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. “Design Lessons from the Fastest Q&A Site in the West.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2011, pp. 2857–2866 (cited on page 18).

Mandelin, David, Lin Xu, Rastislav Bodík, and Doug Kimelman. “Jungloid Mining: Helping to Navigate the API Jungle.” *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, 2005, pp. 48–61 (cited on pages 27, 30).

Markdown Preview Enhanced. <https://github.com/shd101wyy/markdown-preview-enhanced>. Last accessed May 6, 2020 (cited on page 118).

Marks, J., B. Andalman, P.A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. “Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation.” *Proceedings of the International Conference on Computer Graphics and Interactive Techniques*. ACM, 1997, pp. 389–400 (cited on page 53).

Mayer, Mikaël, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. “User Interaction Models for Disambiguation in Programming by Example.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2015, pp. 291–301 (cited on pages 28, 131).

McBurney, Paul W. and Collin McMillan. “Automatic Documentation Generation via Source Code Summarization of Method Context.” *Proceedings of the International Conference on Program Comprehension*. ACM, 2014, pp. 279–290 (cited on page 50).

McDermid, Sean. “Usable Live Programming.” *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 2013, pp. 53–62 (cited on page 49).

MDN contributors. *Code example guidelines*. https://developer.mozilla.org/en-US/docs/MDN/Contribute/Guidelines/Code_guidelines. Last accessed May 6, 2020 (cited on page 20).

Miara, Richard J., Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. “Program indentation and comprehensibility.” *Communications of the ACM* 26.11 (1983), pp. 861–867 (cited on page 10).

Miltner, Anders, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. “On the Fly Synthesis of Edit Suggestions.” *Proceedings of the SIGPLAN Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. Article 143. ACM, 2019, 143:1–143:29 (cited on page 46).

Montandon, João Eduardo, Hudson Borges, Daniel Felix, and Marco Tulio Valente. “Documenting APIs with Examples: Lessons Learned with the APIMiner Platform.” *Proceedings of the Working Conference on Reverse Engineering*. IEEE, 2013, pp. 401–408 (cited on page 24).

Moreno, Laura, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. “Automatic Generation of Natural Language Summaries for Java Classes.” *Proceedings of the International Conference on Program Comprehension*. IEEE, 2013, pp. 23–32 (cited on page 50).

Moreno, Laura, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. “How Can I Use This Method?” *Proceedings of the International Conference on Software Engineering*. IEEE, 2015, pp. 880–890 (cited on pages 24–26).

Murphy-Hill, Emerson and Andrew P. Black. “Refactoring Tools: Fitness for Purpose.” *IEEE Software* 25.5 (2008) (cited on page 46).

Myers, Brad A., Richard McDaniel, and David Wolber. “Intelligence in Demonstrational Interfaces.” *Communications of the ACM* 43.3 (2000), pp. 82–89 (cited on page 28).

Mysore, Alok and Philip J. Guo. “Torta: Generating Mixed-Media GUI and Command-Line App Tutorials Using Operating-System-Wide Activity Tracing.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2017, pp. 703–714 (cited on pages 17, 18, 40–43, 105).

Nasehi, Seyed Mehdi and Frank Maurer. “Unit Tests as API Usage Examples.” *Proceedings of the International Conference on Software Maintenance*. IEEE, 2010 (cited on page 58).

Nasehi, Seyed Mehdi, Jonathan Sillito, Frank Maurer, and Chris Burns. “What Makes a Good Code Example? A Study of Programming Q&A in StackOverflow.” *Proceedings of the International Conference on Software Maintenance*. IEEE, 2012, pp. 25–34 (cited on pages 14–16, 58, 105).

Neal, Lisa Rubin. “A System for Example-Based Programming.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 1989, pp. 63–68 (cited on page 12).

Nielebock, Sebastian, Dariusz Krolkowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. “Commenting source code: is it worth it for small programming tasks?” *Empirical Software Engineering* 24 (2019), pp. 1418–1457 (cited on page 11).

Norcio, A. F. “Indentation, documentation and programmer comprehension.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 1982, pp. 118–120 (cited on page 10).

Nykaza, Janet, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. “What Programmers Really Want: Results of a Needs Assessment for SDK Documentation.” *Proceedings of the International Conference on the Design of Communication*. ACM, 2002, pp. 133–141 (cited on pages 14, 15, 58).

Observable. <https://observablehq.com/>. Last accessed May 6, 2020 (cited on pages 34, 37, 39, 40, 49).

Oezbek, Christopher and Lutz Prechelt. “JTourBus: Simplifying Program Understanding by Documentation that Provides Tours through the Source Code.” *Proceedings of the International Conference on Software Maintenance*. IEEE, 2007, pp. 64–73 (cited on page 38).

Olsen Jr., Dan R. “Evaluating User Interface Systems Research.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2007, pp. 251–258 (cited on page 4).

Oman, Paul W. and Curtis R. Cook. “Typographic Style is More than Cosmetic.” *Communications of the ACM* 33.5 (1990), pp. 506–520 (cited on page 10).

Oney, Stephen and Joel Brandt. “Codelets: Linking Interactive Documentation and Example Code in the Editor.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2012, pp. 2697–2706 (cited on page 58).

Oney, Stephen and Brad Myers. “FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages.” *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2009, pp. 105–108 (cited on page 45).

Oney, Steve, Christopher Brooks, and Paul Resnick. “Creating Guided Code Explanations with chat.codes.” *Proceedings of the Conference on Computer-Supported Cooperative Work and Social Computing*. Article 39. ACM, 2018 (cited on pages 38, 40–42).

Ottenstein, Karl J. and Linda M. Ottenstein. “The program dependence graph in a software development environment.” *ACM SIGPLAN Notices* 19.5 (1984), pp. 177–184 (cited on page 44).

Ou, Jibin, Martin Vechev, and Otmar Hilliges. “An Interactive System for Data Structure Development.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2015, pp. 3053–3062 (cited on page 50).

Parnin, Chris, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. *Crowd Documentation: Exploring the Coverage and the Dynamics of API Discussions on*

Stack Overflow. Tech. rep. Georgia Institute of Technology, 2012 (cited on pages 19, 58).

Parnin, Chris, Christoph Treude, and Margaret-Anne Storey. “Blogging Developer Knowledge: Motivations, Challenges, and Future Directions.” *Proceedings of the International Conference on Program Comprehension*. IEEE, 2013, pp. 211–214 (cited on pages 17, 58, 103, 105).

Parr, T. J. and R. W. Quong. “ANTLR: A Predicated- $LL(k)$ Parser Generator.” *Software—Practice and Experience* 25.7 (1995), pp. 789–810 (cited on page 70).

Pennington, Nancy. “Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs.” *Cognitive Psychology* 19.3 (1987), pp. 295–341 (cited on page 9).

Perez, Fernando and Brian E. Granger. *Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science*. Grant proposal. 2015. <http://archive.ipython.org/JupyterGrantNarrative-2015.pdf> (cited on page 34).

Pirolli, Peter and Margaret Recker. “Learning Strategies and Transfer in the Domain of Programming.” *Cognition and Instruction* 12.3 (1994), pp. 235–275 (cited on page 13).

Pombo, Rodrigo. *Build Your Own React*. Nov. 13, 2019. <https://pomb.us/build-your-own-react/> (cited on page 41).

Pombo, Rodrigo. *Gatsby Waves*. <https://github.com/pomber/gatsby-waves>. Last accessed May 6, 2020 (cited on page 41).

Processing. <https://processing.org/>. Last accessed May 6, 2020 (cited on page 1).

Ragavan, Sruti Srinivasa, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. “Foraging among an Overabundance of Similar Variants.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2016, pp. 3509–3521 (cited on page 87).

Rambally, Gerard K. “The influence of color on program readability and comprehensibility.” *Proceedings of the Technical Symposium on Computer Science Education*. ACM, 1986, pp. 173–181 (cited on page 11).

Ramsey, Norman. “Literate programming simplified.” *IEEE Software* 11.5 (1994), pp. 97–105 (cited on page 32).

Ramsey, Norman and Carla Marceau. “Literate Programming on a Team Project.” *Software—Practice and Experience* 21.7 (1991), pp. 677–683 (cited on page 33).

Ray Wenderlich. <https://www.raywenderlich.com>. Last accessed May 6, 2020 (cited on page 103).

Reinhold, Arnold. *Punched card program deck.agr.jpg*. This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license. 2006. https://en.wikipedia.org/wiki/File:Punched_card_program_deck.agr.jpg (cited on page 22).

Rissland, Edwina L. and Elliot M. Soloway. “Overview of an Example Generation System.” *Proceedings of the AAAI Conference on Artificial Intelligence*. The AAAI Press, 1980, pp. 256–258 (cited on page 27).

Robillard, Martin P. “What Makes APIs Hard to Learn? Answers from Developers.” *IEEE Software* 26.6 (2009), pp. 27–34 (cited on page 58).

Robillard, Martin P. and Robert Deline. “A field study of API learning obstacles.” *Empirical Software Engineering* 16 (2011), pp. 703–732 (cited on pages 14–16, 58).

Rosson, Mary Beth and John M. Carroll. “The Reuse of Uses in Smalltalk Programming.” *ACM Transactions on Computer-Human Interaction* 3.3 (1996), pp. 219–253 (cited on page 13).

RStudio. <https://rstudio.com/>. Last accessed May 6, 2020 (cited on page 37).

Rule, Adam. “Design and Use of Computational Notebooks.” PhD thesis. University of California, San Diego, 2018 (cited on pages 35, 85, 100).

Rule, Adam, Ian Drosos, Aurélien Tabard, and James D. Hollan. “Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding.” *Proceedings of the Conference on Computer-Supported Cooperative Work and Social Computing*. Article 150. ACM, 2018 (cited on pages 36, 37, 39, 86).

Rule, Adam, Aurélien Tabard, and James D. Hollan. *Data from: Exploration and Explanation in Computational Notebooks*. UC San Diego Library Digital Collections. 2018. <https://doi.org/10.6075/J0JW8C39> (cited on page 95).

Rule, Adam, Aurélien Tabard, and James D. Hollan. “Exploration and Explanation in Computational Notebooks.” *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Paper 32. ACM, 2018 (cited on pages 35, 36, 85).

Sacks, Marc. *On-the-Job Learning in the Software Industry: Corporate Culture and the Acquisition of Knowledge*. Quorum Books, 1994 (cited on pages 12, 58).

Sadowski, Caitlin, Kathryn T. Stolee, and Sebastian Elbaum. “How Developers Search for Code: A Case Study.” *Proceedings of the Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 191–201 (cited on page 13).

Sanchez, Huascar, Jim Whitehead, and Martin Schäfer. “Multistaging to Understand: Distilling the Essence of Java Code Examples.” *Proceedings of the International Conference on Program Comprehension*. IEEE, 2016 (cited on page 43).

Schulte, Eric, Dan Davison, Thomas Dye, and Carsten Dominik. “A Multi-Language Computing Environment for Literate Programming and Reproducible Research.” *Journal of Statistical Software* 46.3 (2012), pp. 1–24 (cited on page 34).

Shneiderman, Ben and Richard Mayer. “Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results.” *International Journal of Computer and Information Sciences* 8.3 (1979), pp. 219–238 (cited on page 10).

Shum, Stephen and Curtis Cook. “Using Literate Programming to Teach Good Programming Practices.” *Proceedings of the Technical Symposium on Computer Science Education*. ACM, 1994, pp. 66–70 (cited on page 34).

Siegel, Noah, Nicholas Lourie, Russell Power, and Waleed Ammar. “Extracting Scientific Figures with Distantly Supervised Neural Networks.” *Proceedings of the Joint Conference on Digital Libraries*. ACM, 2018, pp. 223–232 (cited on page 133).

Sillito, Jonathan and Andrew Begel. “App-Directed Learning: An Exploratory Study.” *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE, 2013, pp. 81–84 (cited on pages 12, 14, 15).

Silva, Josep. “A Vocabulary of Program Slicing-Based Techniques.” *ACM Computing Surveys* 44.3 (2012), 12:1–12:41 (cited on page 44).

Sohan, S M, Craig Anslow, and Frank Maurer. “SpyREST: Automated RESTful API Documentation using an HTTP Proxy Server.” *Proceedings of the International Conference on Automated Software Engineering*. IEEE, 2015, pp. 271–276 (cited on pages 28, 30).

Solar-Lezama, Armando. “Program Synthesis by Sketching.” PhD thesis. University of California, Berkeley, 2008 (cited on pages 30, 130).

Soloway, Elliot and Kate Ehrlich. “Empirical Studies of Programming Knowledge.” *IEEE Transactions on Software Engineering* SE-10.5 (1984), pp. 595–609 (cited on page 9).

Sorva, Juha. “Visual Program Simulation in Introductory Programming Education.” PhD thesis. Aalto University, 2012 (cited on page 50).

Sridhara, Giriprasad, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. “Towards Automatically Generating Summary Comments for Java Methods.” *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2010, pp. 43–52 (cited on pages 50, 124, 132).

Sridhara, Giriprasad, Lori Pollock, and K. Vijay-Shanker. “Automatically Detecting and Describing High Level Actions within Methods.” *Proceedings of the International Conference on Software Engineering*. ACM, 2011, pp. 101–110 (cited on page 50).

Sridhara, Giriprasad, Lori Pollock, and K. Vijay-Shanker. “Generating Parameter Comments and Integrating with Method Summaries.” *Proceedings of the International Conference on Program Comprehension*. IEEE, 2011, pp. 71–80 (cited on page 50).

Stack Overflow. <https://stackoverflow.com>. Last accessed May 10, 2020 (cited on page 12).

Stack Overflow Help Center. *How to create a Minimal, Reproducible Example*. <https://stackoverflow.com/help/minimal-reproducible-example>. Last accessed May 6, 2020 (cited on page 20).

Stasko, John, John Domingue, Marc H. Brown, and Blaine A. Price, eds. *Software Visualization: Programming as a Multimedia Experience*. The MIT Press, 1998 (cited on page 50).

Streamlit. <https://www.streamlit.io/about>. Last accessed May 6, 2020 (cited on page 37).

Stylos, Jeffrey and Brad A. Myers. “Mica: A Web-Search Tool for Finding API Components and Examples.” *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2006, pp. 195–202 (cited on page 58).

Subramanian, Siddharth, Laura Inozemtseva, and Reid Holmes. “Live API Documentation.” *Proceedings of the International Conference on Software Engineering*. ACM, 2014, pp. 643–652 (cited on pages 49, 125, 132).

Sulír, Matúš and Jaroslav Porubän. “Generating Method Documentation Using Concrete Values from Executions.” *Proceedings of the Symposium on Languages, Applications and Technologies*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 3:1–3:13 (cited on pages 28, 30, 50).

Suzuki, Ryo. “Interactive and Collaborative Source Code Annotation.” *Proceedings of the International Conference on Software Engineering*. IEEE, 2015, pp. 799–800 (cited on page 38).

Sweigart, Al. *Automate the Boring Stuff with Python*. <https://automatetheboringstuff.com/>. Last accessed May 6, 2020 (cited on page 118).

Tagliaferri, Lisa. *How to Code in Python 3*. DigitalOcean. https://www.digitalocean.com/community/tutorial_series/how-to-code-in-python-3. Last accessed May 6, 2020 (cited on page 119).

Tanimoto, Steven L. “A Perspective on the Evolution of Live Programming.” *Proceedings of the International Workshop on Live Programming*. IEEE, 2013, pp. 31–34 (cited on page 49).

Tanimoto, Steven L. “VIVA: A Visual Language for Image Processing.” *Journal of Visual Languages and Computing* 1.2 (1990), pp. 127–139 (cited on page 49).

Tapp, Riston and Rick Kazman. “Determining the Usefulness of Colour and Fonts in a Programming Task.” *Proceedings of the Workshop on Program Comprehension*. IEEE, 1994, pp. 154–161 (cited on page 11).

Tashtoush, Yahya, Zeinab Odat, Izzat Alsmadi, and Maryan Yatim. “Impact of Programming Features on Code Readability.” *International Journal of Software Engineering and Its Applications* 7.6 (2013), pp. 441–458 (cited on page 9).

Terragni, Valerio, Yepang Liu, and Shing-Chi Cheung. “CSNIPPEX: Automated Synthesis of Compilable Code Snippets from Q&A Sites.” *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 118–129 (cited on page 47).

Thimbleby, H. “Experiences of ‘Literate Programming’ using cweb (a variant of Knuth’s WEB).” *The Computer Journal* 29.3 (1986), pp. 201–211 (cited on page 32).

Thompson, Andrew. *The SSCCE*. <http://www.sscce.org/>. Last accessed May 6, 2020 (cited on page 20).

Tiarks, Rebecca and Walid Maalej. “How Does a Typical Tutorial for Mobile Development Look Like?” *Proceedings of the Working Conference on Mining Software Repositories*. ACM, 2014, pp. 272–281 (cited on pages 17, 18, 103, 105).

Tip, Frank. “A Survey of Program Slicing Techniques.” *Journal of Programming Languages* 3.3 (1995), pp. 121–189 (cited on pages 58, 74).

Tip, Frank, Chris Laffra, Peter F. Sweeney, and David Streeter. “Practical Experience with an Application Extractor for Java.” *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 1999, pp. 292–305 (cited on pages 45, 128).

Toomim, Michael, Andrew Begel, and Susan L. Graham. “Managing Duplicated Code with Linked Editing.” *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2004, pp. 173–180 (cited on pages 48, 117, 128).

Treude, Christoph, Ohad Barzilay, and Margaret-Anne Storey. “How Do Programmers Ask and Answer Questions on the Web?” *Proceedings of the International Conference on Software Engineering*. IEEE, 2011, pp. 804–807 (cited on page 58).

Treude, Christoph and Martin P. Robillard. “Understanding Stack Overflow Code Fragments.” *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2017, pp. 509–513 (cited on pages 19, 58).

Uddin, Gias and Martin P. Robillard. “How API Documentation Fails.” *IEEE Software* 32.4 (2015), pp. 68–75 (cited on page 48).

Uwano, Hidetake, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. “Analyzing Individual Performance of Source Code Review Using Reviewers’ Eye Movement.” *Proceedings of the Symposium on Eye Tracking Research & Applications*. ACM, 2006, pp. 133–140 (cited on page 8).

Vallée-Rai, Raja, Phong Co, Etienne Gagnon, and Sable Research Group. “Soot – A Java Bytecode Optimization Framework.” *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1999 (cited on page 69).

van der Meij, Hans, Joyce Karreman, and Michaël Steehouder. “Three Decades of Research and Professional Practice on Printed Software Tutorials for Novices.” *Technical Communication* 56.3 (2009), pp. 265–292 (cited on page 16).

Van Wyk, Christopher J. “Literate programming: An assessment.” *Communications of the ACM* 33.3 (1990), pp. 361–363 (cited on pages 32, 33).

Victor, Bret. *Explorable Explanations*. 2011. <http://worrydream.com/ExplorableExplanations/> (cited on page 131).

Victor, Bret. *Learnable programming*. 2012. <http://worrydream.com/LearnableProgramming/> (cited on page 49).

VSCode. <https://code.visualstudio.com/>. Last accessed May 6, 2020 (cited on page 118).

WALA. <http://wala.sourceforge.net>. Last accessed May 6, 2020 (cited on page 82).

Wang, Xiaoran, Lori Pollock, and K. Vijay-Shanker. “Automatic Segmentation of Method Code into Meaningful Blocks: Design and Evaluation.” *Journal of Software: Evolution and Process* 26.1 (2014), pp. 27–49 (cited on page 46).

Wattenberger, Amelia. *Interactive Charts with D3.js*. <https://wattenberger.com/blog/d3-interactive-charts>. Last accessed June 4, 2019 (cited on page 41).

Weiser, Mark. “Program slicing.” *Proceedings of the International Conference on Software Engineering*. IEEE, 1981, pp. 439–449 (cited on pages 85, 92).

Weiser, Mark David. “Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.” PhD thesis. University of Michigan, 1979 (cited on page 44).

Weiss, Robert S. *Learning from Strangers: The Art and Method of Qualitative Interview Studies*. The Free Press, 1995 (cited on page 105).

Wiedenbeck, Susan, Vikki Fix, and Jean Scholtz. “Characteristics of the mental representations of novice and expert programmers: an empirical study.” *International Journal of Man-Machine Studies* 39.5 (1993), pp. 793–812 (cited on page 9).

Wightman, Doug, Zi Ye, Joel Brandt, and Roel Vertegaal. “SnipMatch: Using Source Code Context to Enhance Snippet Retrieval and Parameterization.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2012, pp. 219–228 (cited on page 58).

Wikibooks contributors. *A Beginner’s Python Tutorial/Classes*. https://en.wikibooks.org/w/index.php?title=A_Beginner%27s_Python_Tutorial/Classes. Last accessed Sept. 14, 2019 (cited on pages 112, 119).

Wit, Michiel de, Andy Zaidman, and Arie van Deursen. “Managing Code Clones Using Dynamic Change Tracking and Resolution.” *Proceedings of the International Conference on Software Maintenance*. IEEE, 2009, pp. 169–178 (cited on page 48).

Wobbrock, Jacob O. and Julie A. Kientz. “Research Contributions in Human-Computer Interaction.” *ACM Interactions* 23.3 (2016), pp. 38–44 (cited on page 4).

Wu, Yuhao, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. “How Do Developers Utilize Source Code from Stack Overflow?” *Empirical Software Engineering* 24 (2019), pp. 637–673 (cited on pages 12, 14).

Xia, Xin, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. “What do developers search for on the web?” *Empirical Software Engineering* 22 (2017), pp. 3149–3185 (cited on page 13).

Yang, Di, Aftab Hussain, and Cristina Videira Lopes. “From Query to Usable Code: An Analysis of Stack Overflow Code Snippets.” *Proceedings of the Working Conference on Mining Software Repositories*. ACM, 2016, pp. 391–401 (cited on page 19).

Ying, Annie T. T. and Martin P. Robillard. “Selection and Presentation Practices for Code Example Summarization.” *Proceedings of the International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 460–471 (cited on page 17).

Yoon, YoungSeok and Brad A. Myers. “An Exploratory Study of Backtracking Strategies Used by Developers.” *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE, 2012, pp. 138–144 (cited on page 35).

Zeller, Andreas and Ralf Hildebrandt. “Simplifying and Isolating Failure-Inducing Input.” *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200 (cited on page 45).

Zhang, Tianyi, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. “Are Code Examples on an Online Q&A Forum Reliable? A Study of API Misuse on Stack Overflow.” *Proceedings of the International Conference on Software Engineering*. IEEE, 2018, pp. 886–896 (cited on pages 19, 47).

Zhang, Tianyi, Di Yang, Crista Lopes, and Miryung Kim. “Analyzing and Supporting Adaptation of Online Code Examples.” *Proceedings of the International Conference on Software Engineering*. IEEE, 2019, pp. 316–327 (cited on pages 14, 47).

Zhang, Xiangyu, Neelam Gupta, and Rajiv Gupta. “Pruning Dynamic Slices with Confidence.” *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, 2006, pp. 169–180 (cited on page 44).

Zhang, Xiong and Philip J. Guo. “DS.js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science.” *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2017, pp. 691–702 (cited on page 49).

Zwicky, F. “The Morphological Approach to Discovery, Invention, Research and Construction.” *New Methods of Thought and Procedure: Contributions to the Symposium on Methodologies*. Springer, 1967, pp. 273–297 (cited on page 51).