# Authoring Multi-Stage Code Examples with Editable Code Histories

**Shiry Ginosar, Luis Fernando De Pombo, Maneesh Agrawala, Björn Hartmann**
Computer Science Division, University of California, Berkeley
shiry@cs.berkeley.edu, lfdepombo@berkeley.edu, maneesh@cs.berkeley.edu, bjoern@cs.berkeley.edu

## ABSTRACT

Multi-stage code examples present multiple versions of a program where each stage increases the overall complexity of the code. In order to acquire strategies of program construction using a new language or API, programmers consult multi-stage code examples in books, tutorials and online videos. Authoring multi-stage code examples is currently a tedious process, as it involves keeping several stages of code synchronized in the face of edits and error corrections. We document these difficulties with a formative study examining how programmers author multi-stage code examples. We then present an IDE extension that helps authors create multi-stage code examples by propagating changes (insertions, deletions and modifications) to multiple saved versions of their code. Our system adapts revision control algorithms to the specific task of evolving example code. An informal evaluation finds that taking snapshots of a program as it is being developed and editing these snapshots in hindsight help users in creating multi-stage code examples.

### Author Keywords

Examples; tutorials; programming; editable histories

### ACM Classification Keywords

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

## INTRODUCTION

Programmers can find technical material needed for specific programming tasks by searching for community-generated content online. While novice programmers may mostly look for syntactic help, more advanced developers often need to acquire strategies for combining smaller library functions and APIs to create full programs. One way to learn such strategies is through *multi-stage code examples* which present multiple versions of a program where each stage increases the complexity of the overall example [33].

Indeed, teaching materials often use multi-stage code examples such as the one presented in Figure 2 (on the next page). This simplified example introduces elements of an interactive drawing program one-by-one. The first stage shows how to draw a single line. The second shows how to set drawing parameters such as canvas size and color. The third stage adds interactivity by drawing lines to the cursor position. Finally, the last stage clears the screen so only a single line is drawn while the mouse is dragged.

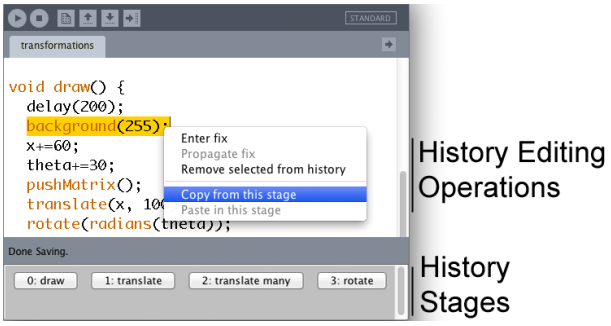


**Figure 1. We introduce an editor that facilitates the construction of multi-stage code examples through a set of history editing operations and the management of history stages.**

Creating multi-stage code examples requires keeping track of multiple versions of code and the edits between them. In a formative study where we observed coders as they created such examples, we found that authors have difficulty planning example stages from beginning to end. They often go back and forth in stage history, making changes over multiple versions. For example, if the lines in an example designed for print are difficult to see because they are too thin (Figure 2), the author would have to modify stages 2-4 to consistently update the thickness parameter.

Revision control systems like git [13] or Subversion [36] provide tools for controlling multiple versions of code. However, their user interface is insufficient to support the specific task of authoring code examples. For instance, after committing several revisions of a program, there is no easy way to add a block of code to the existing sequence of revisions without requiring the user to intervene and resolve conflicts. A tool that assists users with such editing operations would ideally be integrated into an Integrated Development Environment (IDE) to make for a seamless development and editing experience (Figure 1). Our formative study suggests that this type of editor should support an addition operation so that authors can make changes to multiple versions of code. It should also enable authors to remove or fix existing blocks of code throughout the history of example stages. Ultimately, for seamless authoring of multi-stage code examples, pro-


*UIST'13*, October 8–11, 2013, St. Andrews, United Kingdom.
ACM 978-1-4503-2268-3/13/10.
http://dx.doi.org/10.1145/2501988.2502053

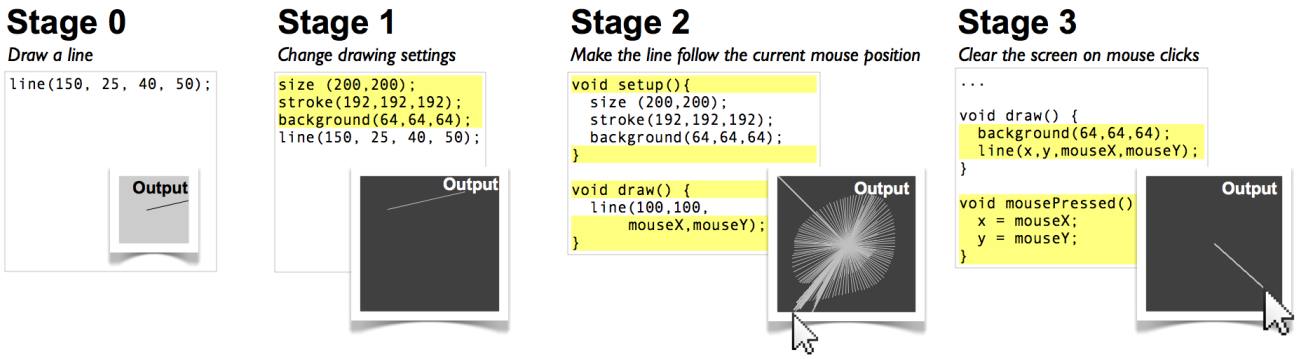


**Figure 2. In multi-stage examples, each stage builds upon the existing code and adds new functionality (highlighted). Stage 0 shows how to draw a single line. Stage 1 shows how to set drawing parameters such as canvas size and color. Stage 2 adds interactivity by drawing lines to the cursor position. Finally, Stage 3 clears the screen so only a single line is drawn while the mouse is dragged. (Adapted from the Processing.org).**

grammers need to be able to edit the history of their code, propagating changes across stages.

We present an authoring tool that facilitates the creation of multi-stage code examples (Figure 1). Our tool extends a standard IDE, and captures a rich representation of code into an underlying version control system. It introduces IDE interaction techniques that enable example authors to edit the example history by propagating edits across stages. The techniques we use are based on differencing algorithms that identify the point in history where a change was introduced, and the most likely location of code blocks in other stages of the example program. Some of these algorithms are included in revision control tools and others, like the ability to add code to previous revisions, extend the functionality of these tools.

Authors can use our system to convert a set of code stages to multi-stage code examples that can be directly consumed by learners. We take inspiration from code examples in books, tutorials and websites, and demonstrate the generation of one output format as a proof of concept. In this HTML format we highlight the differences in the lines of code between stages and we include the running output of each stage as it is viewed directly in the web page. Additionally, learners can easily navigate between example stages.

We conducted an informal study of our tool with seven participants and found that they frequently saved snapshots of their code as stages while writing examples (up to 11 stages per example). Additionally, they successfully edited across stages to modify or fix their examples.

## BACKGROUND

Several different types of learning materials use multi-stage code examples to teach strategies of program construction. For example, programming books and online tutorials often start with a simple construct and develop a more complicated example over several chapters. Many video tutorials begin with an empty project and develop a working program. However, these often do not demonstrate discrete, working code in separable stages [37]. In this section, we review people's programming learning strategies to motivate the importance of supporting the authoring of multi-stage examples.

### How Programmers Use Code Examples

Programmers face different barriers when trying to write code in a new environment [20]. *Use barriers* represent difficulties in determining how to work with a programming interface. *Coordination barriers* concern the rules for combining individual building blocks to achieve complex behaviors. Both barriers are well addressed by concrete code examples. Coordination strategies in particular can be taught by introducing many examples of programs as they are being developed [33]. By following these examples, a learner can observe how others combine smaller structures to create a working program.

Users are not inclined to read lengthy training materials; their particular goals and concerns often differ from the careful ordering of chapters in a training procedure [6]. This trend also holds for programmers: when programmers find tutorials online they often experiment with the included code samples rather than take the time to read the accompanying explanation. Moreover, programmers often begin adapting code before they completely understand it [4]. When a developer feels that the information he already collected is sufficient to complete a given task, the developer may drop out of the comprehension process and switch to implementing a solution with found information [21]. In line with these observations, Minimalist Instruction Theory [6] calls for designing materials as small, self-contained units representing real tasks that can be read in any order. Multi-stage examples share similarities with early Minimalist systems as they break the experts' program creation process into discrete chunks. Each of these chunks has a well defined effect on the output of the program enabling out-of-order access and exploration by the learners. By exporting multi-stage examples to interactive formats like HTML they allow for the presentation of output formats similar to Carroll's ViewMatcher that presented example programs in an interactive tool together with their output [7] and to Victor's designs that make the state of the program visible [40].

## RELATED WORK

Our work builds on two primary areas of prior work; tools for authoring and using code examples, and tools for visualizing and editing operation histories.

### Authoring and Using Code Examples

Like our work, JTutor [22] focuses on code changes over time; it permits record-and-replay of code edits, but does not offer the history editing we provide. The commercial Codeplayer [37] records and replays coding sessions at the character level. Like single-take screencasts, such performances require careful planning on the author's part. Our history editing techniques enable example authors to make modifications to different stages at any point.

Augmented examples provide additional information beyond a static view of a piece of source code. JTourBus [28] lets developers record annotated "tours" through a project to highlight important code locations to others. WebEx allows teachers to produce example code with line-wise explanations [5]. HyperSource [17] and Codetrail [14] save relevant URLs along with source code as a form of design rationale. Our work instead focuses on capturing additional temporal information on how code evolves.

Several researchers have investigated how to help learners find, extract or synthesize code examples [3, 8, 24, 25, 34]. Other systems aim to make example code reuse easier [29, 30, 42]. In contrast, we focus on example authoring.

Researchers have investigated how to author examples and tutorials for other domains besides programming. Many studies concentrate on image manipulation and 3D modeling applications because of the complexity of their interfaces. Several systems generate tutorials for such interfaces from demonstrations [1, 10, 15]. Given a structured representation of a step-by-step procedure, such systems can also observe a user's actions and synchronize the display of instructions with those actions [1, 2, 31]. Demonstrations in 3D Modeling or photo manipulation can comprise hundreds of operations and therefore call for intelligent techniques for aggregating or navigating steps [11, 15, 16]. The dependence or independence of different actions in an editing sequence can be captured by revision control functions adapted to image manipulation [9]. Our system is unique with respect to these prior works as none of these systems are designed to build multi stage code examples.

### Editable Histories

Editable code histories in our work are closely related to editable graphical histories, which provide authors with tools to modify prior document states. Researchers built such systems for vector graphics [23, 35], photo manipulation [9], information visualization [18], and other domains. Common actions supported by these tools, like ours, are navigation to prior states and editing a prior state to propagate changes forward in time. We also allow editing later states and propagating changes back. More importantly, our work differs in that the final product of our tool is not a single document but rather a sequence of documents over time. In this sense, the collected history is the final artifact.

The ability to edit multiple code stages is related to Linked Editing, a technique to link duplicate code blocks in a program and edit them simultaneously [38]. However, unlike multi-stage code examples, the Linked Editing model does not capture a sequential history of modifications.

The Git rebase command also enables programmers to apply changes and additions of code to a chain of commits [13]. However, our system can better resolve conflicts when we propagate the addition of code to past stages since (1) we assume there is one author who is guided to only merge in additive changes and (2) we let the user impose location constraints on both ends (start and end) of the propagation.

## FORMATIVE STUDY

To better understand the challenges programmers face when producing multi-stage code examples, we conducted a formative study with seven participants. We asked each participant to produce a multi-stage code example that describes how to program the classic Pong game in Processing, a Java variant [32]. We chose Processing as the language of our study for several reasons. First, Processing is designed for creating images, animations and interactive graphical sketches. Graphical programs have visual output and allow for at least one straightforward way to define different stages of program construction as points at which the visual output changes. Second, simple programs can be expressed very concisely in Processing, while complex projects remain possible.

Participants in the formative study were students at our institution already familiar with Processing. We gave them a program editor that we linked to a revision control system, so that we could simulate a tool that makes it easy to snapshot any stage of the development of a program by committing a revision to the system. We provided participants with paper printouts of the snapshots of their code so that they could view their snapshots side by side and thereby simulate the ability to switch between stages in the editor.

Our participants all made high-level plans to subdivide the study task into meaningful example stages. In practice, they created an average of 4.4 stages ($\sigma = 1.9$) as they developed their program. However, they did not always follow their initial plans. At times the plans contained mistakes and they had to re-plan the stages. In other cases they had to fix minor code errors in earlier stages. All participants faced multiple instances where they realized that earlier implementation decisions had to be modified in order to proceed. On average, participants indicated that they would have edited previous revisions an average of 1.6 times ($\sigma = 0.89$) while programming Pong.

### Types of Modifications to Previous Revisions

We noted two types of modifications that users indicated that they wanted to perform on earlier stages of the code.

#### *Modifications for Pedagogical Reasons*

The goal of some modifications was to construct a flow of stages that learners could easily follow. For instance, one participant defined the ball velocity in the first stage as a per-second velocity. However, when developing later stages she decided that it would be more instructive to refer to velocities as per-frame velocities, as these require fewer unit conversions when writing frame-based rendering routines. She then

wished to change the definition of the velocity variable and add an extra frame rate variable to her first stage.

*Modifications as a Result of an Error*

Other modifications were the result of errors in the code that programmers did not notice at first, but later wanted to fix. For example, one participant noticed that she had accidentally swapped the window width and height variables throughout her program. She wished to modify all the earlier versions of the code in which the variables were switched.

### The Need for Editable Histories

Making such modifications using a regular IDE would require propagating the change to all relevant stages manually. In some cases, this process of change propagation was complex enough that some participants remarked that they would have preferred to jettison much of their work and restart from an earlier point. In contrast to editing code for the purpose of the final program, participants had to not only think about refactoring the current state of the code, but also check that the flow across stages remained consistent.

In essence, while participants were able to formulate high-level plans, detailed decisions were made just-in-time based on existing code. Producing multi-stage examples is thus fundamentally a non-linear process that requires making modifications to multiple stages of the code project.

### Design Guidelines

Based on our formative study we devised the following design guidelines to support multi-stage editing inside an IDE.

*Enable Easy Addition of Stages*

Programmers should be able to capture snapshots of the current state of their code without interfering with their coding workflow. When we provided participants with this ability, they took full advantage of it and saved multiple snapshots.

*Enable Easy Navigation and Execution of Stages*

Programmers have to be able to access all stages in the editor with minimal effort. Participants frequently went back to saved snapshots of their code in order to remind themselves of how they got to the current state. A natural extension to this guideline is to allow programmers to execute the code as it exists in each stage in order to see the resulting output.

*Enable Edits of Any Stage*

Programmers must be able to modify any stage of their code. We distinguish between three different types of edits that we observed in the formative study:

- **Propagate code fixes across stages:** allowing errors such as the switched height and width to be fixed.
- **Add code across stages:** allowing omissions such as variable definitions to be included where they belong.
- **Remove code across stages:** allowing participants to change their minds in hindsight about existing code such as the introduction of the ball in the very first stage.

*Enable the Removal of Stages*

Based on the experience of at least one participant it seems worthwhile to be able to remove a saved stage such that the previous stage flows directly to the next without the intervening one. This would have allowed participants to delete stages they later determined had no educational value.

## AUTHORING A MULTI-STAGE CODE EXAMPLE

To illustrate how these design guidelines may be expressed in a tool for editing code histories consider the following example. The task is to show the drawing primitives necessary to build a program that rolls an object off the screen using translation and rotation (Figure 3).

### Adding a Stage to the History

A programmer begins by making a high level plan and breaking down the task she is working on into small testable units. The programmer chooses to start by drawing a circle (Figure 3A). Once this code is tested the programmer saves a snapshot of the current state of the code. A dialog box lets the programmer specify the title for the stage (Figure 4A).

The programer adds a second stage where she translates the circle to a new location and a third stage where she animates the repeated translation of the circle (Figure 3B). We borrow Kurlander's approach [23] and add a button in the history panel for every new stage (Figure 4B). The programmer can click this button to view a previous stage.

### Modifying Code Across Stages

A programmer can modify code in any stage in two ways. They can make changes to the latest stage and selectively propagate them to past stages, or they can make changes to a past stage and selectively propagate them to later ones.

*Fixing Blocks of Code Across Stages*

The programmer next wants to rotate the object as it rolls off the screen. However, because of the circle's rotational symmetry she decides to change her object to a square (Figure 3C). She can either fix her choice of a shape in the stage where the circle was introduced and propagate the fix to future stages, or fix it in her current code and propagate the fix to all previous stages. In this case, she chooses the latter. She selects and highlights the line creating the circle and right clicks on the **Enter fix** option (Figure 4C). She then edits the code in place to change the shape to a square and propagates the change back to all other stages using **Propagate fix** (Figure 4D). Finally, she adds code to rotate the square as it is translated, to simulate rolling off the screen (Figure 3D).

*Adding Blocks of Code Across Stages*

Viewing the current output of her program, the programmer realizes that the demonstration of rotation is not clear enough, since the square also has a rotational symmetry. She decides to transform the square into a more complex shape – in this case a robot face. Thus she adds shapes representing the eyes and mouth of the robot to all stages of the example.

Adding code to a different stage is a form of a copy-paste operation where a block of text is copied from the present and is pasted in a past stage, or vice versa. Once it is pasted, it is also included in all intermediate stages. The programmer decides to add the drawing of facial features in the current code. She then selects and highlights the code she just added, right

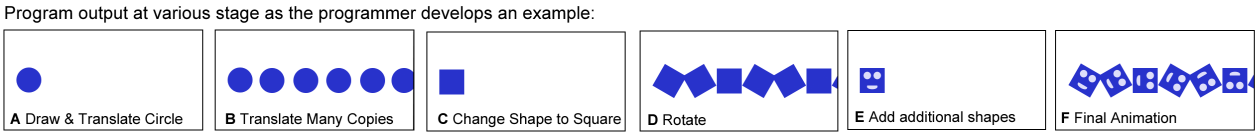


**Figure 3. A programmer takes different steps during the development of the example, reverting and adding to earlier decisions.**

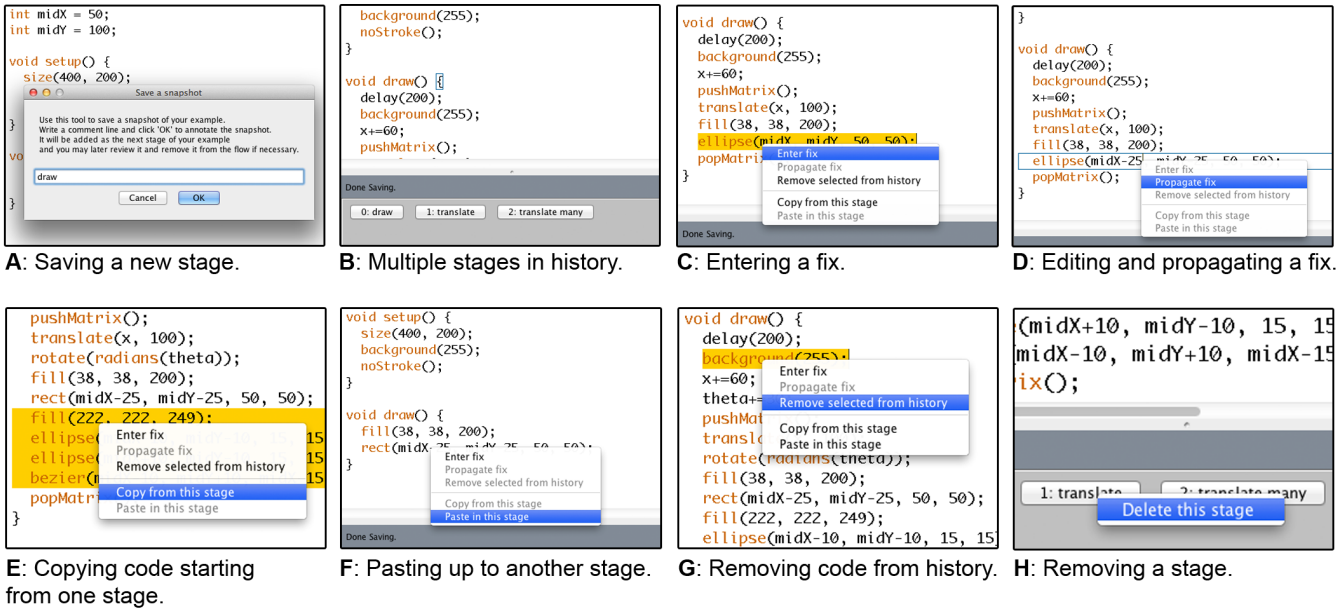


**Figure 4. The Multi-stage example editor offers functions to manage code stages and functions to perform edits across multiple stages.**

clicks and chooses the **Copy from this stage** option (Figure 4E). Navigating to the very first stage, the programmer points to the end of the line after which she would like to paste and chooses the **Paste in this stage** option (Figure 4F). The copied code is pasted in the past stage and propagated to all the stages in between. As a result the robot is drawn in all stages. If the programmer attempts to paste in a location that conflicts with the location of the code in the current stage the operation fails and an error message is displayed. This could occur, for instance, if the programmer were to copy code from the bottom of the file at the current stage and paste it at the very top of the file in a past stage.

*Removing Blocks of Code from History*

The programmer next decides that she would like to better emphasize the result of each rotation operation for educational purposes. She therefore removes the line that clears the window each time the drawing function is called. She selects and highlights the line that clears the background, and selects the **Remove selected from history** option (Figure 4G). The deleted code is removed from the stage in which it was introduced and onwards. Alternatively, the programmer could have navigated to the stage where the code was introduced, removed it from there and propagated the fix to later stages.

### Presenting Negative Effects of Edits

As the number of stages in the example increases, the programmer is more likely to make mistakes when editing across stages. For example, removing a declaration of a variable from all stages may break a past stage that includes code to increment that variable. Before performing any change that affects multiple stages, our tool simulates the changes and re-compiles each affected stage. If any compilation fails the programmer is notified that her change would break the corresponding stage in history. The programmer can then decide whether to approve or cancel the edit. While this check can detect compile-time problems it cannot detect runtime errors.

### Removing a Stage from the Code History

On a final read-through of her complete example, the programmer decides that the stage where she translated the shape by a fixed amount is superfluous. She right-clicks on the stage's button and chooses the **Delete this stage** option (Figure 4H). This option deletes the snapshot of the state of her code at that point of history. However the edits from the deleted stage are still reflected in subsequent stages.

### Publishing Examples

The purpose of writing multi-stage code examples is to produce materials that learners can easily consume. Important design considerations for such materials are how to enable *navigation* between the various stages, and how to present *changes* between stages. In our implementation, when the programmer is finished authoring a code example, she may publish it as an interactive web page. Learners can then navigate between stages using a draggable control. In order to convey the differences between stages, we highlight the changed lines between subsequent stages and include the running output of the program at each stage (Figure 5).

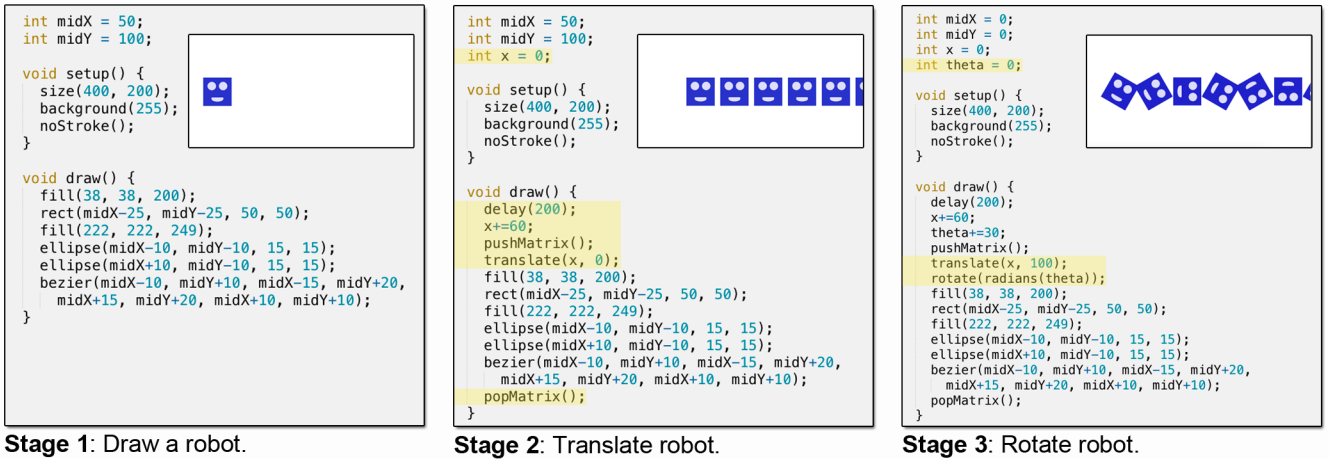


**Figure 5. The exported learning materials enable viewers to navigate stages and see changes in source as well as program output. The export format was reformatted here for the sake of presentation.**

The example format presented here is merely a proof of concept. Given the rich representation of the program history collected by our system we believe that many other output formats may be possible. We leave the problem of finding the optimal format for displaying multi-stage code examples as future work.

## IMPLEMENTATION

We wrote our authoring tool in Java as an extension of the open source Processing IDE [32]. However, the principles behind our algorithms are not tied to any particular language or editor. In addition to the IDE's representation of the program, we maintain a separate representation of the code in a git repository [13]. This allows us to use revision control operations as the basis for our history editing algorithms.

### Adding a Stage to the History

Example stages are snapshots of code in time. Saving such snapshots are the bread and butter of revision control systems and we therefore delegate snapshotting to git [13]. When the programmer opens a new file to author a Processing program (called a "sketch"), a repository is initialized in the sketch folder to track a copy of the code in a text file. Each time the programmer takes a new snapshot of her code for a stage, its current state is committed to git as a new revision.

### Modifying Code Across Stages

We support several types of edits across sequences of stages.

#### *Fixing Blocks of Code Across Stages*

In order to allow a programmer who is working on the current stage to fix an existing block of code (Figure 6A, code highlighted in orange) in all stages where it appears (Figure 6A, code highlighted in yellow), we must first locate the stage where the author introduced it. This is the earliest place where we should propagate the fix. Similarly, when a programmer directly edits a past stage and propagates code fixes to later stages, we must locate the last stage where the code appears. We look for exact copies of the code (ignoring whitespace) because modified lines of code may have different semantics. Propagating changes to stages where the author has modified the relevant block of code would require

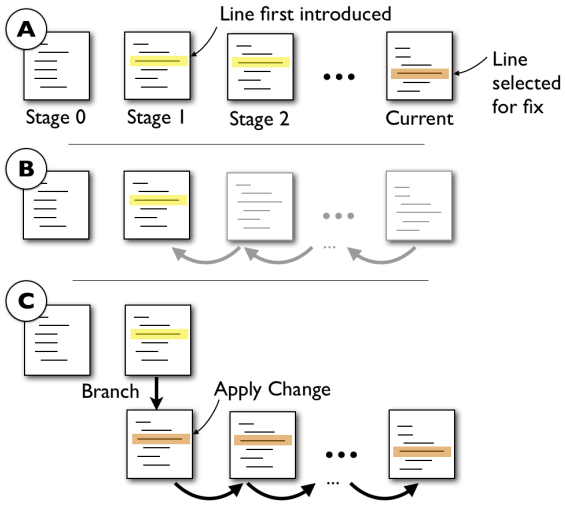


**Figure 6. Fixing existing code. A. When the fix is propagated to the past, the revision where the code first appeared is located using git pickaxe. B. The code is rolled back. C. A new brach is created where the fix is applied, and the code is rolled forward to the current state.**

additional user input to disambiguate how changes should be propagated to different lines, as our tool cannot decide this automatically for the users.

In our system we use the git pickaxe utility to identify the stages where each line in a block of code that is to be fixed across stages appears. Given each line to fix in the code, the pickaxe utility returns all revisions that changed that line (Figure 6A). If the fix is propagated to a past stage, the system rolls back to the first revision where the line appeared (Figure 6B). The system then creates a new branch where the line is fixed and recommitted (Figure 6C). The fix is rolled forward to the current revision by applying all subsequent changes from the original branch. The operation is similar when the fix is propagated to a later stage.

Removing code is just a special case of fixing as it can be expressed as a fix where the new version is an empty line.

*Adding Blocks of Code Across Stages*

Adding a block of code across a sequence of stages requires additional constraints from the user: the first and last stage where the addition should occur, and the exact lines in these stages where the code should be added. These constraints are needed in operations such as taking a new line $X$ that is between lines $A$ and $B$ in the present and adding it at a past stage before $A$ or $B$ were introduced. In this case, there may be several possible locations of line $X$ in the past stage and we use the provided location to resolve the ambiguity.

We allow addition of new lines of code to a sequence of revisions ranging from some past revision up to the present state. This supports both the case where the user is copying from the present and pasting to a past stage and the case where the user is editing a past stage and propagating to the present. The difference lies only in the user interface. In order to implement this forward propagation, we introduce a new algorithm for adding blocks of code to code histories through a modified three-way merge process.

Generally, three-way merge algorithms [12] reconcile independent changes in two files under the assumption that they share the third file as a common origin. If the changes are compatible and would not conflict by altering the same lines, then a new file will be produced. If not, then the algorithm attempts to identify the cause of the failure. Users of revision control systems, where two or more programmers may independently change the current revision, commonly experience such failures. In our tool, copying and pasting a chunk of code from one stage to another can be thought of as a similar situation. While the paste location is given by the user, we must determine how this new change can be merged into the existing flow of changes from stage to stage. For example, in Figure 7 if we are pasting changes into Stage 1 (Figure 7A) then the first two files to be merged would be Stage 1 after pasting ("Stage 1P") and "Stage 2", and the common origin would be "Stage 1" (Figure 7B).

Our implementation uses the three-way merge capability of the UNIX diff3 tool [12]. When merging two sets of changes into an original there is only a small set of cases where there is no ambiguity concerning the ordering of lines and therefore a guaranteed correct merge. General purpose tools like diff3 are conservative in what they will accept as a non-conflicting merge and report a conflict whenever there is an ambiguity. To shield developers from having to manually resolve such conflicts, we can take advantage of user-provided position constraints and the knowledge that all edits are additions.

We automatically resolve the conflicts that the three-way merge identifies. Unlike the general three-way merge problem, we have an advantage when propagating pasted code forward. Here the original stage with the paste applied may only contain additional lines of code (Figure 8) as fixes and deletions are covered by the previous techniques. Our merge therefore resolves the conflict by accepting all changes to the original code (stage 1 in Figure 8) introduced by the next stage (e.g., the removal of line A in Figure 8, stage 2), and only using pasted additions (line B in Figure 8, stage 1P) in the resulting merge.

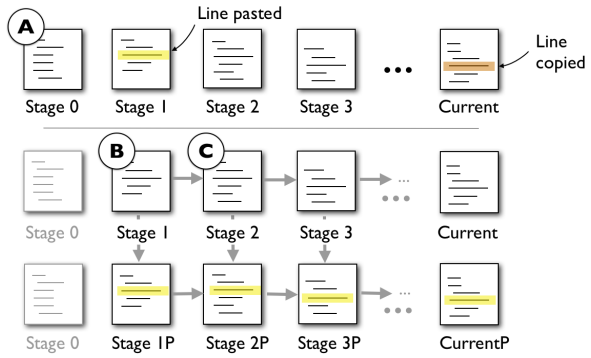


**Figure 7. Adding a block of code to past stages using a copy-paste operation. A. The programmer wants to introduce a new block of code at an earlier stage. She copies it from the current local changes and pastes it in the past stage. B. The first step of propagating the paste to all stages in between. At every step we perform a three-way merge between an example stage (Stage 1), the stage with the pasted code (Stage 1P) and the next stage in history (Stage 2). The result of the merge is a modification of Stage 2 containing the pasted code (Stage 2P). C. In the next step stages 2, 3 and the previous result 2P are merged.**

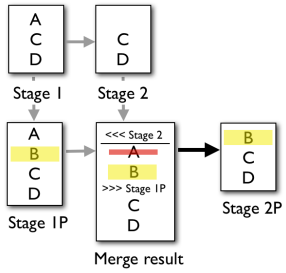


**Figure 8. We resolve conflicts by only looking at the original file with the code pasted in it (stage 1P) for the location in which to paste the code. All other changes from stage 1P (like adding line A) are discarded.**

We can recursively apply this process until we reach the current stage. In each step of the recursion, the result of the previous merge (e.g. "Stage 2P" in Figure 7C) serves as one of the three files to merge. To check if the propagation succeeded we compare the result of the final merge against the correct state of the current stage. We consider all possible places to paste the additional code in each stage and ultimately select the sequence of placements that results in the correct state of the code, if such a selection exists. It may not always be feasible to find the right sequence of paste positions, for instance if the user copied the last line of a stage and pasted it as the first line of the previous stage.

### Presenting Negative Effects of Edits

In order to warn the user of potential negative effects of her actions, we optimistically apply the edit and ensure that all stages still compile. This background compilation test is performed on a copy of the code file and using a clone of the git repository. If all stages of the edited example compile successfully, the edit is applied to the code. If the edit breaks a stage, we inform the user of the failure and undo the edit. Furthermore, the system directs the user to the first stage that did not compile when attempting to apply the edit.

### Removing a Stage from the Code History

At times, programmers take snapshots of key points in the construction of the code that later no longer seem essential for the sake of the example. When a previously saved stage is removed we roll back all stages up to and including it. Next we apply the changes from the stage to be removed and the next one then commit these changes together. Lastly, we roll forward to the current revision. The final artifact in this case does not change, only the sequence of stages.

### Publishing Examples

We chose to publish finished code examples in HTML as a demonstration of one possible output format. The log extracted from git is parsed by a Python script and converted to HTML using the Jinja2 templating library [19].

Since the published examples are designed to be consumed by learners, we allow easy navigation between stages using an extension of the Tangle kit [41], a Javascript library for creating reactive documents that offers draggable controls to show or hide objects from view on a web page [39].

We employ two strategies in order to help learners understand what changed from stage to stage. First, we highlight the changed lines between stages when moving from one to the other using JQuery utilities. Second, we use Processing.js to embed in the page the running interactive output of the program at every stage, so that even without reading the code itself the effect of each stage is immediately visible.

## EVALUATION

To evaluate the effectiveness of our system for authoring multi-stage code examples we conducted an informal use study. We asked participants to use our tool to modify an existing multi-stage example and to generate a code example of their own.

### Participants

We recruited seven participants, all graduate students at our institution, with prior expertise in Processing. The six Computer Scientists and one Mechanical Engineer, ages 22 to 28, all rated themselves as expert programmers. Six were male; one female. Two of the participants took part in our formative study. Only one participant self reported to be an expert in tutorial writing; the rest had at most passing knowledge.

### Methodology

Each two hour session in our lab consisted of a warm-up task, two experimental tasks, and a post-study survey. In the warm-up task, we first introduced participants to the features of our system using step-by-step instructions for constructing a simple code example. The goal of this task was to acquaint participants with the propagation features of our system. A preliminary version of our system that only included one propagation direction for each kind of operation was tested. This initial version also did not include background compilation to show potential problems of edits.

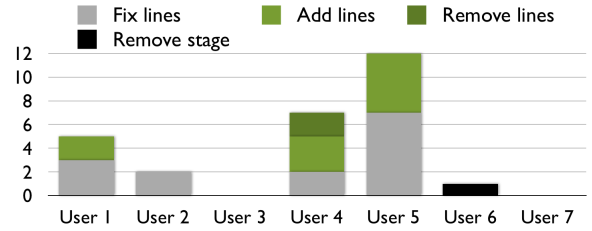


**Figure 9. The number of history editing operations used by the evaluation study participants during the multi-stage example generation task.**

In the second task, we asked participants to make various changes to a given multi-stage code example for the game of Pong. The example exhibited several poor design and didactic choices that we asked participants to improve, such as the proportion of the sizes of the ball and paddle, the contrast of colors, and the inclusion of superfluous example stages. Participants edited the example such that the changes were reflected throughout the history of its construction using the operations of our system.

In the third task, we asked each participant to create their own multi-stage code example using our system. This task was designed to mimic real-world use of the system. To check that our propagation tools are applicable in a variety of contexts, we assigned a different programming assignment to each participant. We asked some of the participants to write games, such as Snake and a raindrop catching game. Others built tools, such as a color picker. Others created motion simulations of objects and particles.

### Results

All participants successfully completed the warm-up task. They all used all available features during the second modification task as participants successfully translated our edit suggestions into appropriate history editing operations. Following our modification suggestions, participants used each feature once, except for fixing code which they used 4 times.

When authoring their own code examples during the third task, all participants structured their code in a modular, multi-stage fashion, and saved between 3 and 11 stages ($\mu = 6.4, \sigma = 2.5$). While programming, all participants frequently navigated back to previous stages to remind themselves of the state of the code at that time, either by reading or executing the code, and often by doing both. Given the variety in the examples they authored and the individual differences in the writing process between people, it was not surprising that different participants used a different subset of the modification features. Participants used an average of 3.86 history editing features during the generation task (Figure 9). In the third task users 3 and 7 did not use any features of our tool because one implemented his example correctly on the first trial, and the other was not able to construct a full example in the allotted time.

In order to assess whether the user view of the available operations matched the system view [26], participants rated whether the effects of each operation matched their expectations and the average match rating was relatively high. Detailed results are presented in Figure 10.

#### *Strengths*

When asked which operation was the most useful to them in authoring a multi-stage code example, participants' answers

covered the whole range of available operations. This result suggests that all the operations are necessary. All participants attested that they would be very likely to use this tool for authoring multi-stage examples.

**The ability to save snapshots and navigate between them:** In our post-study survey, participants noted that the main advantage of using our tool for authoring multi-stage code examples was the ability to easily save snapshots of their code. Participants also mentioned the value in being able to switch rapidly between the stages and re-run the code at each stage. One participant noted that being able to save snapshots helped him remember to partition his task into concrete subtasks.

**The ability to edit past stages:** Participants liked the fact that they didn't have to make all design decisions from the very start since they could make changes and reflect them in previous versions on-the-fly. Thus, they could focus on the main goal for each stage, rather than worry about each little detail. One person noted that he felt he could almost write the code as he would for his own purposes, while producing a tutorial at the same time.

**The ability to publish examples:** An additional advantage reported by participants is the ability to publish their examples directly into a format that is consumable by readers without any post-processing other than possibly adding explanatory text. During the modification task participants used publication in a way that we did not anticipate. When trying to understand the example given to them, participants exported the example to HTML format in order to view the differences between each stage and observe the running output. We interpret this usage as a sign that our output format is valuable for sharing examples between authors as well as with learners.

### *Shortcomings*

Participants identified possible improvements of features.

**Support directly editing past stages:** In the version of our tool used in the evaluation, all our modification operations required the user to modify code in the present and then propagate the changes back to the past. This metaphor worked for some users, but not for others who would rather have the ability to edit the code in past stages directly and propagate the edits forward (Cognitive Science has established that different people prefer different metaphors for reasoning about time [27]). During the study, these users found it more natural to reason about modifications in a chronological order. They tended to switch back through earlier stages each time they propagated a modification, effectively checking that the system behaved as expected. In fact, our underlying architecture allows for edit propagation in any direction. Following the study, we realized that we should enable edits to past stages - adding it only required a simple change to the UI.


**Figure 10. Post-experiment questionnaire results. Error bars indicate $\frac{1}{2}$ standard deviation in each direction.**

**Need for improved interaction techniques for editing:** Two users would have preferred not to indicate when they are about to enter code fixes. Instead, they wanted the system to infer the fix from the local changes in the code. Other users identified shortcomings in the usability of the current features, specifically a lack of visual feedback. For instance, one user suggested that all local unsaved changes in an example stage be highlighted to make it easier to remember what may be propagated back. Another user suggested highlighting all allowable places in the past where a copied piece of code could be pasted. Users also expressed interest in the ability to rename a stage, the ability to add a new stage between two existing past stages, and the ability to split their current local changes into two separate stages.

## LIMITATIONS

Our approach has some inherent limitations that limit its generality; our prototype's implementation places additional restrictions on users.

Our matching algorithm does not have any semantic understanding of the code – all operations are performed at the character or line-level although the UI allows for aggregate operations on blocks of code. As a result, our modification algorithm will not attempt to propagate fixes to any lines that do not match the original line being modified by the user. Future work could investigate if comparisons at the abstract syntax tree (AST) level could relax this restriction.

We assume that stages represent a linear progression of code development. Alternatively, programmers may wish to develop modular examples where viewers can enable or disable different aspects of an example interactively. This investigation is left to future work.

## CONCLUSION

We presented a set of interaction techniques and underlying algorithms that permit programmers to propagate code changes to multiple history stages. These techniques were motivated by a formative study that found that programmers needed to make such modifications to fix mistakes and adjust didactic strategies when writing multi-stage examples. An informal evaluation of our initial implementation for the Processing IDE showed that programmers can successfully use our techniques to author and modify code across stages. An open empirical question for future work is how to best present the resulting examples to different communities of learners online. More generally, we are interested in pursuing additional work on creating tools that benefit both authors as well as consumers of learning materials for programming.

## ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE 1106400 and NSF Award IIS-1149799.

## REFERENCES


1. Bergman, L., Castelli, V., Lau, T., and Oblinger, D. Docwizards: a system for authoring follow-me documentation wizards. In *Proceedings of UIST*, ACM (2005), 191–200.
2. Berthouzoz, F., Li, W., Dontcheva, M., and Agrawala, M. A framework for content-adaptive photo manipulation macros: Application to face, landscape, and global manipulations. *ACM Transactions on Graphics. 30*, 5 (Oct. 2011), 120:1–120:14.
3. Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S. R. Example-centric programming: integrating web search into the development environment. In *Proceedings of CHI*, ACM (2010), 513–522.
4. Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., and Klemmer, S. R. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of CHI*, ACM (2009), 1589–1598.
5. Brusilovsky, P. Webex: Learning from examples in a programming course. In *WebNet'01* (2001), 124–129.
6. Carroll, J. M. *The Nurnberg funnel: designing minimalist instruction for practical computer skill*. MIT Press, 1990.
7. Carroll, J. M., Singer, J. A., Bellamy, R. K. E., and Alpert, S. R. A view matcher for learning smalltalk. In *Proceedings of CHI*, ACM (1990), 431–437.
8. Chang, K. S.-P., and Myers, B. A. Webcrystal: understanding and reusing examples in web authoring. In *Proceedings of CHI*, ACM (2012), 3205–3214.
9. Chen, H.-T., Wei, L.-Y., and Chang, C.-F. Nonlinear revision control for images. In *ACM SIGGRAPH 2011 papers*, ACM (2011), 105:1–105:10.
10. Chi, P.-Y., Ahn, S., Ren, A., Dontcheva, M., Li, W., and Hartmann, B. Mixt: automatic generation of step-by-step mixed media tutorials. In *Proceedings of UIST*, ACM (2012), 93–102.
11. Denning, J. D., Kerr, W. B., and Pellacini, F. Meshflow: interactive visualization of mesh construction sequences. In *ACM SIGGRAPH 2011 papers*, ACM (2011), 66:1–66:8.
12. UNIX diff3. http://www.unix.com/man-page/FreeBSD/1/diff3/.
13. Git Distributed Version Control System. http://git-scm.com/.
14. Goldman, M., and Miller, R. C. Codetrail: Connecting source code and web resources. *Journal of Visual Languages and Computing 20*, 4 (2009), 223–235.
15. Grabler, F., Agrawala, M., Li, W., Dontcheva, M., and Igarashi, T. Generating photo manipulation tutorials by demonstration. *ACM Transactions on Graphics 28*, 3 (July 2009), 66:1–66:9.
16. Grossman, T., Matejka, J., and Fitzmaurice, G. Chronicle: capture, exploration, and playback of document workflow histories. In *Proceedings of UIST*, ACM (2010), 143–152.
17. Hartmann, B., Dhillon, M., and Chan, M. K. Hypersource: bridging the gap between source and code-related web sites. In *Proceedings of CHI*, ACM (2011), 2207–2210.
18. Heer, J., Mackinlay, J. D., Stolte, C., and Agrawala, M. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1189–1196.
19. Jinja2. http://jinja.pocoo.org/docs/.
20. Ko, A. J., Myers, B. A., and Aung, H. H. Six learning barriers in end-user programming systems. In *Proceedings of VLHCC*, IEEE (2004), 199–206.
21. Ko, A. J., Myers, B. A., Coblenz, M. J., and Aung, H. H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering 32*, 12 (Dec. 2006), 971–987.
22. Kojouharov, C., Solodovnik, A., and Naumovich, G. Jtutor: an eclipse plug-in suite for creation and replay of code-based tutorials. In *Proceedings of eclipse technology eXchange*, ACM (2004), 27–31.
23. Kurlander, D., and Feiner, S. Editable graphical histories. In *IEEE Workshop on Visual Languages* (oct 1988), 127 –134.
24. Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. Jungloid mining: helping to navigate the API jungle. In *Proceedings of PLDI*, ACM (2005), 48–61.
25. Mooty, M., Faulring, A., Stylos, J., and Myers, B. A. Calcite: Completing code completion for constructors using crowds. In *Proceedings of VLHCC*, IEEE (2010), 15–22.
26. Norman, D. A., and Draper, S. W. *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., 1986.
27. Nunez, R. E., Motz, B. A., and Teuscher, U. Time after time: The psychological reality of the ego- and time-reference-point distinction in metaphorical construals of time. *Metaphor and Symbol 21*, 3 (2006), 133–146.
28. Oezbek, C., and Prechelt, L. Jtourbus: Simplifying program understanding by documentation that provides tours through the source code. In *ICSM*, IEEE (2007), 64–73.
29. Omar, C., Yoon, Y., LaToza, T. D., and Myers, B. A. Active code completion. In *Proceedings of ICSE*, IEEE Press (2012), 859–869.
30. Oney, S., and Brandt, J. Codelets: linking interactive documentation and example code in the editor. In *Proceedings of CHI*, ACM (2012), 2697–2706.
31. Pongnumkul, S., Dontcheva, M., Li, W., Wang, J., Bourdev, L., Avidan, S., and Cohen, M. F. Pause-and-play: automatically linking screencast video tutorials with applications. In *Proceedings of UIST* (2011), 135–144.
32. Reas, C., and Fry, B. Processing: programming for the media arts. *AI Society 20*, 4 (Aug. 2006), 526–538.
33. Robins, A., Rountree, J., and Rountree, N. Learning and teaching programming: A review and discussion. *Computer Science Education 13* (2003), 137–172.
34. Stylos, J., Faulring, A., Yang, Z., and Myers, B. A. Improving api documentation using api usage information. In *Proceedings of VLHCC*, IEEE (2009), 119–126.
35. Su, S. L., Paris, S., Aliaga, F., Scull, C., Johnson, S., and Durand, F. Interactive visual histories for vector graphics. Tech. Rep. MIT-CSAIL-TR-2009-031, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, June 2009.
36. Apache Software Foundation. Subversion. http://subversion.apache.org//.
37. The code player. http://thecodeplayer.com/.
38. Toomim, M., Begel, A., and Graham, S. L. Managing duplicated code with linked editing. In *Proceedings of VLHCC*, IEEE (2004), 173–180.
39. Victor, B. Explorable explanations. http://worrydream.com/ExplorableExplanations/.
40. Victor, B. Learnable programming. http://worrydream.com/LearnableProgramming/.
41. Victor, B. Tangle. http://worrydream.com/Tangle/.
42. Wightman, D., Ye, Z., Brandt, J., and Vertegaal, R. Snipmatch: using source code context to enhance snippet retrieval and parameterization. In *Proceedings of UIST*, ACM (2012), 219–228.