UNFOLD: Enabling Live Programming for Debugging GUI Applications

Ruanqianqian (Lisa) Huang
University of California, San Diego
La Jolla, CA, USA
r6huang@ucsd.edu

Philip J. Guo
University of California, San Diego
La Jolla, CA, USA
pg@ucsd.edu

Sorin Lerner
University of California, San Diego
La Jolla, CA, USA
lerner@cs.ucsd.edu

***Abstract*—Debugging GUI applications is challenging because of the difficult-to-debug state changes caused by asynchronous event handling and user interactions. Live programming, a paradigm where programmers continuously see real-time traces of every execution step as they edit the code, is promising for this context, as it automates the visualization of state changes upon inputs to the program. This paper explores how to design a live programming experience for debugging GUI applications and studies the effects of live programming in this context. Through a formative design exploration, we derive three core concepts for enabling live programming in debugging GUI applications: a UI states timeline, connections between the UI and the code, and automated event recording. We implemented these concepts in UNFOLD, a live programming environment for JavaScript-based GUI applications. A within-subject study with 12 participants shows that, with UNFOLD, participants locate bugs faster in tasks amenable to live programming, leverage liveness when debugging, and deem the tool helpful and easy to use.**

***Index Terms*—live programming, GUI applications, debugging, event handling**

I. INTRODUCTION

Debugging can be time-consuming and frustrating for programmers [1]–[3]. In the context of GUI (Graphical User Interface) applications, debugging is even more challenging [4] due to the use of event handlers to react to user interactions. This asynchronous event handling of user interactions could result in difficult-to-debug state changes and thus complicate the debugging task, making program behavior difficult to replicate or too complex to reason about [5].

Live programming is a paradigm where the programmer receives immediate feedback or *liveness* on *every step* of the execution continuously as the code is updated [6]–[8], typically in the form of runtime values [6], [9] or traces [10] on each program change. It is lightweight, requiring no additional effort from the programmer besides an input to the program to obtain such information. Prior work has shown that live programming encourages more frequent testing [11], facilitates finding bugs [12] and program comprehension [13]–[15], and lowers the cognitive load in validating AI-generated code [16]. In the context of debugging, which is an iterative activity with frequent edit-run cycles [17], liveness could facilitate this workflow by reducing potential interruptions. More importantly, the run-time traces generated by liveness imply the *order* of state changes, which is precisely what the programmer might need for reasoning about program behavior. As GUI applications might involve complex state changes, live programming could bring potential benefits to resolving unexpected behavior in these applications.

To meet the promises of live programming for GUI applications, we must overcome several notable challenges. First, since event handling is the key to making GUI applications interactive, the live feedback in the context of event handling must be provided with the user interactions that trigger the required events, and these user interactions must persist throughout the live feedback. Second, live programming for GUI applications must provide run-time traces that present several interconnected concepts during execution: user interactions, program data, and changes to the GUI. Most importantly, the traces must elucidate the *ordering* between all of these. Indeed, the application of live programming to GUI applications and their event handling has yet to be fulfilled. Furthermore, there is still a need for a comprehensive solution that connects user interactions, runtime data and GUI changes, all in a live way.

To address these challenges, this paper explores the design of a live programming experience for debugging GUI applications. To better understand the needs of programmers in this space and guide the design, we conducted a needfinding study. From this study we derive three core concepts for enabling liveness in this setting: a UI states timeline, UI-code correspondences, and automated event triggering. Together, these concepts allow the programmer to see a detailed visualization that connects code, data, and UI changes, all within the context of user interactions, all while being updated in a live way on each code change, and with no additional tool configuration or context switching between the editor and the debugger.

As a proof-of-concept of this idea, we created and evaluated the usage of UNFOLD, a live programming environment for web-based GUI applications that shows a timeline of changed UI states, code that caused those changes, and automatic replays of prior user interaction traces upon each code edit. UNFOLD aligns with the notion of liveness [7] because it visualizes every step of visual changes as a result of user interaction traces and auto-refreshes these visualizations upon edits to the code. By always showing changed UI states and code that caused them, UNFOLD eliminates the need to repeatedly restart execution or replicate user interaction inputs.

Example Usage Scenario. Alice is a front-end web engineer

1943-6106/24/$31.00 ©2024 IEEE
DOI 10.1109/VL/HCC60511.2024.00041

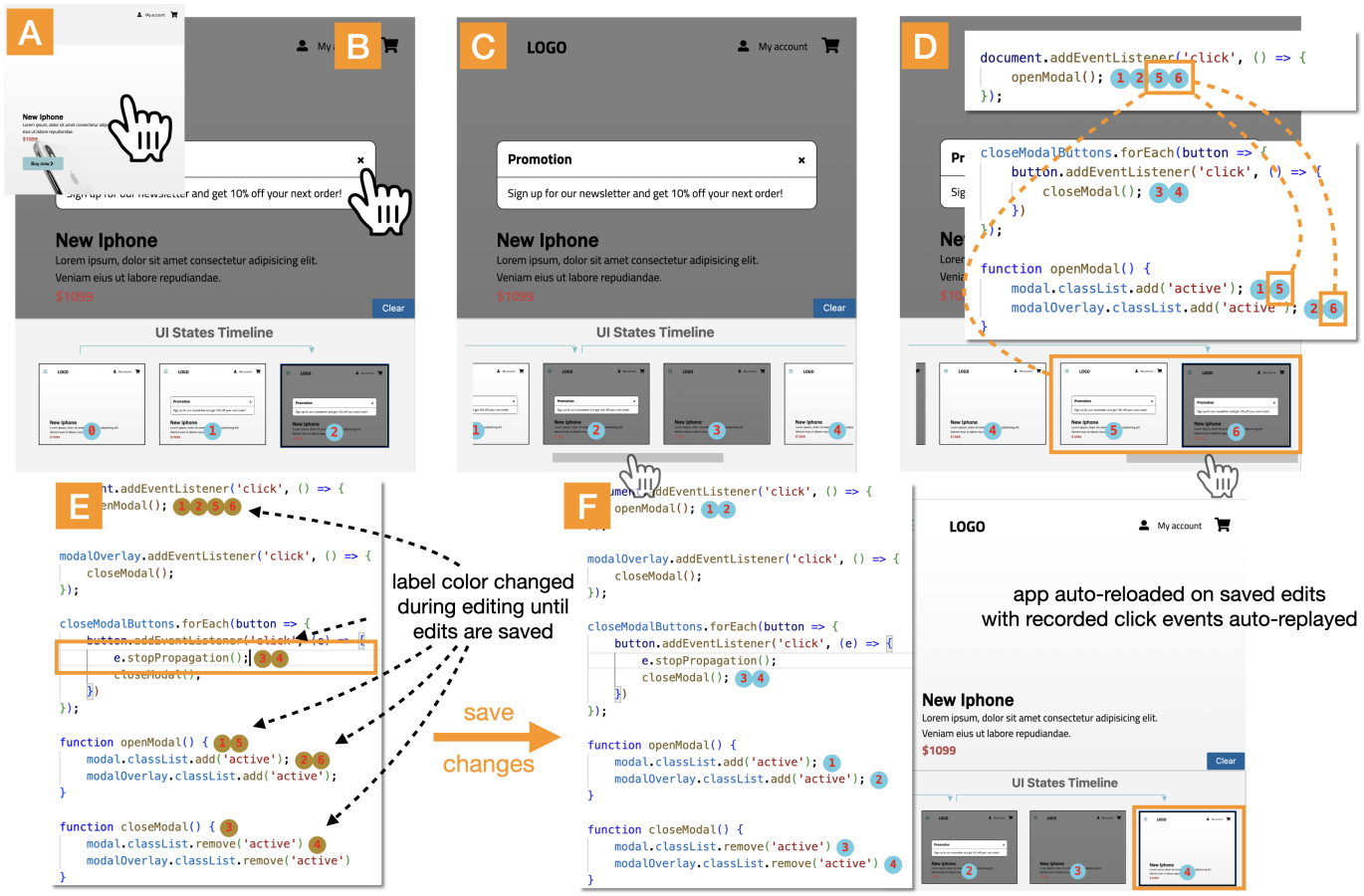

Fig. 1. UNFOLD enables users to do *live programming* by interacting normally with the app, examining the timeline of UI states, seeing how buggy UI states connect to code, editing and saving the code, and immediately seeing updated UI states.

trying to debug some faulty code that her colleague wrote. Her team is building a landing page for a shopping website. When the user clicks on a particular product description, the web app pops up a modal dialog box to show a promotion for it, and that pop-up should disappear when they hit the "close" button. However, there is a bug somewhere as the pop-up does not disappear when the "close" button is pressed. Fig. 1 shows how Alice can find and fix this bug using UNFOLD by interacting normally with the webpage without repeating her interactions or restarting the execution.

(A) She clicks on a product on the webpage like a user does.

(B) Immediately she sees a pop-up of the promotion as expected. The UI States Timeline at the bottom of UNFOLD shows the initial page UI state labeled with a (0) circle, then two intermediate UI states labeled with (1) and (2) showing the pop-up appearing (1) and the page background darkening (2). This looks fine to her, so she clicks the "close" button.

(C) On the page the pop-up did not disappear as expected, which indicates a bug. But in the UI States Timeline, states labeled (3) and (4) show that the pop-up did disappear and is no longer visible in the UI, which seems strange to Alice.

(D) Inspecting further, when she scrolls to the right in the UI States Timeline, it shows that the pop-up did indeed disappear but *later came back* in the subsequent states labeled (5) and (6). By examining the code annotated with labels shared by the unwanted UI states (5) and (6), she realizes that the "click" event handler that shows the pop-up modal dialog box was called at a time when she did not expect it to be.

(E) Re-inspecting the structure of the DOM, Alice sees that this is a bug related to event bubbling [18]. The `click` event received on the "close" button bubbled up to the entire page (a parent DOM node), causing its own `click` event handler to execute. Alice quickly fixes this bug by adding a `stopPropagation()` call to the beginning of the `click` event handler for the "close" button to stop event bubbling.

(F) As soon as Alice saves her changes, UNFOLD live re-runs her code with the user interactions from her prior session, and now the pop-up disappears as she expects. She further confirms from the auto-refreshed UI states visualization that the application now behaves exactly as she expects.

Without UNFOLD, Alice would have needed to insert print

logging or breakpoints, edit her code, reload the page, click to interact with it, inspect the logging or debugger output (which may contain too little or too much output), and iterate until she finds the bug. She might put logging or breakpoints in the wrong places at first and then need to re-run and repeat her page interactions a few times. A single button click may not seem so tedious, but a realistic debugging scenario involves a much longer chain of complex user interactions that need to be manually repeated on each trial. UNFOLD eliminates this tedium by showing UI state changes and live re-running the edited code with user interactions from prior runs.

To assess how UNFOLD works in practice, we ran a comparative within-subjects study, where 12 programmers debugged event handlers in GUI applications using UNFOLD versus standard browser developer tools. We found that, when using UNFOLD, programmers located some GUI application bugs faster, used runtime information more when debugging, and deemed the tool usable and helpful.

This paper's main research contributions are:

- A new application of live programming to debugging GUI applications, which shows state changes and connections to the code that caused those changes.
- A prototype of this concept, UNFOLD, and a comparative user study showing its potential effectiveness for debugging.

II. Related Work

A. Live Programming

Live programming or *liveness* allows programmers to modify a program while receiving immediate feedback on how the edits affect *every step* of the execution [6], [7], [19], typically in the form of runtime values [6], [9] or traces [10]. It has been applied to settings such as general-purpose languages [9], [20], [21], data science [13], [14], [22], and physical computing [11], [15], [23]. Prior work has found via qualitative evidence the potential benefits of liveness for program comprehension [14], [15] and debugging [12], [24], [25]. A recent mixed-method study shows its potential help for validating AI-generated code [16].

To our knowledge, the application of live programming to developing or debugging GUI applications remains under-explored. The closest to our work is RDE [26], which enables liveness for event handling. However, RDE restricts the program to having one rendering function and no function values in the global state, creating no code-output correspondences, which are key to live programming. Our work differs from RDE in that we impose no restrictions on how the code should be written and visualize code-output correspondences. Another relevant work is Theseus [10], which live-visualizes runtime traces of code execution for JavaScript code. Our work adopts the in-situ visualization for runtime traces in Theseus but further does the following that Theseus lacks: automating user interactions and visualizing the *temporal order* of both UI changes and data changes.

Our work UNFOLD is motivated by the promise that live programming holds for understanding and debugging GUI applications. Our work further extends prior live programming work in two aspects. First, while prior live programming work uses input that is already part of the program to enable liveness, UNFOLD automatically records and replays inputs *outside of the program* itself, *i.e.*, user interactions. Second, while prior work presents runtime traces on the level of runtime data only, UNFOLD consolidates user interactions, runtime data, and GUI changes in one live visualization.

In addition to live programming, *live coding* is a related yet different line of work [6]. Although similarly named and also automatically updating program output upon code edits, live coding differs from live programming by showing the *final* state only, while live programming continuously visualizes program behavior *in full* including intermediate states [6], [7]. Existing live coding tools for developing GUI applications [27]–[32] are inspirations for our work regarding designing the experience of immediacy.

B. Debugging and Comprehension Tools for GUI Applications

There is a rich line of debugging systems targeting GUI applications without live programming. Amulet [4] is among the first steps towards usable debugging tools for GUI apps, which supports changing data values at runtime. Timelapse explains changes of a specified DOM element [5], which, similar to UNFOLD, automatically captures user interactions and replays execution. Clematis captures application behavior to a timeline of episodes and shows the event trace, execution trace, and DOM mutations for each [33]. Doppio visualizes high-level correspondences between callback methods and GUI changes in Android apps [34]. Log-It augments print debugging for web apps [35]. Most recently, Hypothesizer enables debugging web applications with a hypothesis-based approach [36]. However, none of these tools provide direct connections from UI states back to code upon edits, which UNFOLD implements.

Finally, there are systems for inspecting and understanding the complex behavior of GUI applications, including web applications [37]–[40] and Snap! applications [41]. Although these tools do not enable code edits or incremental testing of the edits, which are core to debugging, they inspire several techniques we use to enable live programming for GUI applications. Specifically, we were inspired by the code-to-UI connections in these systems to support understanding of GUI behavior. We further extended these systems by (1) reporting errors when they occur in a Timeline area, (2) auto-recording event inputs while the user interacts with the app, and (3) providing live feedback on how code edits affect every step of the visual evolution and code-UI connections in the recorded event context.

III. Formative Study and Design Goals

Designing a live programming environment for debugging GUI applications remains under-explored. To establish design goals for this setting, we ran a formative observational study with five programmers debugging JavaScript web event handling code similar to our example from Fig. 1.

A. Formative Study Protocol

While we detail our formative study in a replication package[1], we summarize the protocol below.

Participants. We recruited five expert JavaScript programmers through Twitter whose experience with JavaScript ranged from 6 to 12 years. The participants had JavaScript experience in at least three settings out of the following: industry, academic research, open-source, school projects, and personal projects.

Procedure. We conducted a contextual inquiry [42] via video conferencing, where each participant spent up to 60 minutes debugging one behavior of a given application and, if time permitted, resolving other bugs. Throughout the study, we noted down actions participants performed to achieve certain goals. Such actions came from direct observations, participants' explanations via think-aloud when programming, and the interviewer's hypotheses confirmed/corrected by the participants during and after the programming session.

Setup and Tasks. Participants programmed in their preferred editors and were allowed to search the web for programming help as they normally would. We randomly assigned the five participants to work with one of the two broken web applications taken from real-world examples: a todo list app and a web-based spreadsheet, both of which involve synchronizing UI state changes with an underlying data model.

B. Design Goals

We briefly summarize our observations and the three design goals (**D1**, **D2**, **D3**) that we derived from the study, which can alleviate problems our participants encountered when using existing tools.

D1: Non-Interruptive Tracing of UI State Changes. Four out of our five participants chose not to use the debugger that comes with browser devtools [43]–[45] to inspect the web app's behavior because, as one said, *"it is interruptive."* They opted for print statements instead because the browser debugger would prevent them from seeing a bigger picture of the execution without breakpoint pauses. The problem is exacerbated in GUI apps as debuggers interrupt tracing the UI states caused by event handlers, which to our participants was the most important. Existing debuggers force users to step through long, unrelated UI state changes with unavoidable pauses, which is interruptive and, as one participant stated, *"does not simulate how the application naturally responds to a triggered event."* Thus, our first design goal is non-interruptive tracing of state changes, which liveness precisely supports.

D2: Connecting Output Changes to the Corresponding Code. All five participants inserted print statements via `console.log()` to verify the execution of event handlers and determine the sequence of code execution, data changes, and UI state changes. However, they had to switch among the UI, code, and console to obtain such information. Console logs also could not immediately show the order of code execution and data changes unless carefully constructed and formatted. Participants wished to see the connection between code and changes in data and GUI output, especially the *order* of these changes, but they currently need to rely on manually sifting through textual logs to obtain such information indirectly. Such a need is indeed what liveness implies.

D3: Automated Event Triggering. All participants found it tedious to manually trigger an event sequence repeatedly (e.g., via mouse clicks) when debugging event handlers, especially when the debugging took numerous rounds of editing. Also, changing one segment of event handling code might unexpectedly affect the behavior of another segment. When editing code for event handlers, users yearned for a way to *"preview the effect of code changes on the execution of these handlers before committing to such changes."* Live programming would free users from repeating event inputs and restarting the execution via its immediate feedback on code edits.

IV. Design & Implementation of Unfold

We implemented a prototype live programming environment for web-based GUI applications called Unfold, which is an extended version of Visual Studio Code [46] that displays the user's code, an iframe containing the live webpage rendered from that code, and a UI States Timeline. Unfold is available on GitHub[2]. We revisit the components of Fig. 1 and use Fig. 2 to describe the design and implementation of Unfold based on the three design goals (**D1**, **D2**, **D3**) derived from our formative study in Sec. III.

A. Live Feedback on Code Edits

Unfold provides immediate feedback on how a code change affects the application's behavior whenever that change is saved, which supports both ***D1: Non-interruptive Tracing of UI State Changes*** and ***D3: Automated Event Triggering***. Furthermore, if the user has unsaved changes, Unfold reminds them of the mismatch between code and output via the UI-to-Code Connector Labels (Sec. IV-C), the color of which changes from the usual blue to dark brown to indicate that the code and output is out of sync.

Liveness is enabled via event recording. Indeed, the user can interact with their web app's UI in the iframe through mouse clicks as usual, and Unfold will automatically record these clicks and replay them each time the code is changed and saved. Unfold records and replays one event sequence at a time to visualize the UI states created by that sequence, which is displayed in the UI States Timeline (Sec. IV-B). To input a new sequence, the user clicks the "Clear" button above the UI States Timeline (Fig. 3).

Our prototype records only click events because they are the most common in GUIs, but our implementation allows easy extensions to other DOM events for record-replay. Unfold captures all click events occurred within the rendered web app at the right of the interface (the blue rectangle in Fig. 2) and the event targets. To auto-replay the recorded events, Unfold uses a "Headless Debugger" (Fig. 2), which consists

[1]Formative study replication package: https://bit.ly/unfold-formative.

[2]Artifact: https://github.com/rlisahuang/unfold-tool

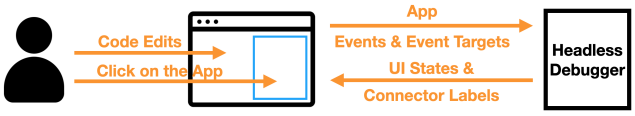

Fig. 2. Architecture of UNFOLD: the recorded `click` events (and the targets on which they occurred) and the HTML file of the application are passed to a "Headless Debugger" for obtaining all the UI states and connector labels info for visualization. The "Headless Debugger" consists of a headless Chrome instance for rendering the HTML and the Chrome debugger protocol that simulates the events in the headless Chrome.

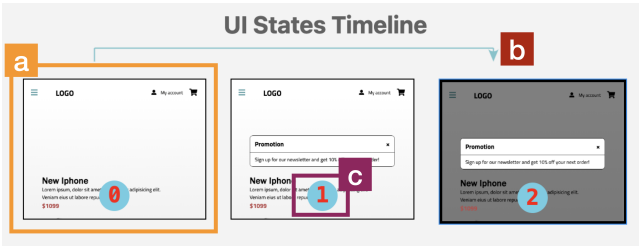

Fig. 3. A zoomed-in view of the UI States Timeline in Fig. 1-A: The Timeline shows all UI states created by each recorded click event. (a) Each state can be expanded to see details; (b) an arrow connects the starting and ending states caused by this click event; (c) shows an intermediate UI state with a label (1), which also appears in the IDE over the line of code that caused this state change (see Fig. 1).

of the Chrome debugger protocol [43] and a headless Chrome instance. Specifically, users interact with the app via the iframe in UNFOLD, which is further connected to a headless Chrome instance in the background that renders the same app and obtains information following the procedure below. First, to obtain event target information for replaying each event, UNFOLD calculates the CSS selector of the event target by tracing down the DOM tree of the web app. This approach allows us to replay events without relying on the dimensions of the web app or the resolution of the computer screen. Then, it sets one breakpoint for each event via the Chrome debugger protocol [43]. Finally, it simulates the event on the target via the headless Chrome with the debugger enabled and records related runtime information (more in Sec. IV-B).

B. UI States Timeline

Fig. 3 zooms in on the UI States Timeline from Fig. 1-A, which implements ***D1: Non-interruptive Tracing of UI State Changes***. After the user clicks on the web app, whenever there is a visual change to the entire UI, a UI state is created. As such, the UI States Timeline shows the starting UI state and the subsequent UI states that have been created since the click event in the order of time. The user can further select a UI state in the timeline to see its larger view. For example, in Fig. 1, the last UI state in the timeline is selected, with its larger view projected in UNFOLD.

UNFOLD records snapshots of UI states by pausing execution at each click event handler and single-stepping with the Chrome debugger protocol through every line of the handler, including nested function calls. Every time the debugger is paused (due to the previously set event breakpoints described in Sec. IV-A), UNFOLD captures the `outerHTML` and a full UI screenshot of the web app as displayed in UNFOLD's iframe viewport (to match what users see). Then it diffs consecutive screenshots on the pixel level and discards duplicates (which indicates that this line of code did not change the UI in a visible way). This also means that UNFOLD does not capture off-screen changes. At the end of the click handler, all distinct screenshots appear as separate UI states created by running that handler and rendered in the UI States Timeline. When a UI state in the timeline is selected, its larger view is rendered via the corresponding `outerHTML` to enable interactions. If a line of code runs an animation (e.g., fade-in), then it will record the UI state at the start and end of that animation. Note that if the click handler runs many lines of code, this approach can become slow (due to the single-stepping), and we have not yet optimized for performance; but in practice, developers often write click handlers to be short-running so that the UI updates quickly to foster interactivity [47].

Our implementation may seem similar to the MutationObserver API that detects DOM changes. However, MutationObserver constantly observes changes beyond the visual changes and would be an overkill for our case. More importantly, our approach reports both the existence and *the lack of* changes more intuitively by capturing all (or the lack of) UI changes occurring as the click event handler executes through screenshot-diffing.

Above the sequence of UI states in Fig. 3, an arrow (Fig. 3-b) is shown for *each* click event – the arrow spans all UI states that the UI passes through when that event is handled. In this example it shows the starting UI state of the webpage (labeled with a (0)), the pop-up modal dialog appearing (state (1)), and then the background of the page darkening (state (2)). Event arrows are blue when they start and end in different-looking UI states, and orange otherwise, which shows that the recorded event did not visibly change the UI.

Our formative study participants (Sec. III) all found print statements to be useful. Thus, in addition to inspecting UI states in this timeline, UNFOLD allows the user to also use console print logging for inspecting internal program state. But instead of logging to a separate console, UNFOLD embeds the logs in the GUI so that each log output results in a visible UI state change, thus enabling users to examine log output directly within the UI States Timeline. This mechanism enables users to reason about the sequence of internal data changes along with visible UI changes together in one timeline.

C. UI-to-Code Connector Labels

The numbered circles shown on UI states (Fig. 3-c) visually connect those UI states to code execution, which addresses ***D2: Connecting Output Changes to the Corresponding Code***. Specifically, if the script that has led to the creation of the UI states is in an open buffer, then all lines that have resulted in UI changes will be annotated with the connector labels of the corresponding UI states. They are blue if the code is in sync with the visual information rendered by UNFOLD or dark brown if there are unsaved changes.

The semantics of these numbered labels is defined as follows: Suppose we have a UI state annotated with a label i, and a line of code annotated with the same label. This means two things: (1) right before the i-labeled line of code started executing, the state of the UI was the one annotated with $i-1$ in the UI States Timeline; (2) right after the i-labeled line of code finished executing, the state of the UI was the one annotated with i in the timeline. These labels use numerical ordering to visualize control flow in the source code at a glance and connect them to UI state changes. Fig. 1 shows more examples of labels next to lines of code.

To annotate code lines with their respective connector labels, whenever the debugger is paused when replaying events, besides capturing UI states (Sec. IV-B), UNFOLD also obtains the line numbers of all statements on the call stack that are from local scripts loaded along with the web application. It then indexes the UI states and associates the recorded line numbers with the UI state indices.

V. EVALUATION: COMPARATIVE USER STUDY

How does liveness affect debugging in GUI apps, including locating and fixing bugs? How does live programming compare to a traditional debugging paradigm in this setting? To investigate these questions, we ran a within-subjects study where each participant debugged two GUI applications using UNFOLD and Firefox DevTools [44] in a randomized order, and we collected task performance, debugging behavior, and user impressions of each tool. This section summarizes our evaluation study protocol; full details of the tasks and procedure can be found in our replication package[3].

Participants. We recruited 12 adults (6 female) with 1 to 15 years of JavaScript experience and 5 to 15 years of programming experience via personal contacts, social media, and our institution. There was no overlap between the formative and comparative user study participants.

Tasks. Each participant debugged two GUI applications, Memory Game (M) and Calculator (C)[4], and Fig. 4 illustrates these applications in UNFOLD. Each comes with two bugs: one directly related to event-driven DOM/CSS manipulations, a critical part of JavaScript event handling [33], and the other caused by data changes. Here are the bugs by application (M and C) labeled with bug type:

M1. [CSS] A card disappears when trying to flip it over.
M2. [Data] Off-by-one error to flip back a pair of flipped cards.
C1. [DOM] Pressing any value key shows `undefined`.
C2. [Data] Pressing "=" after an operation makes no calculation.

The replication package details the location of each bug.

Conditions. Participants edited code in the VSCode IDE [46] in the study with two different debugging setups on top: live programming (UNFOLD) and Firefox DevTools [44] (CONTROL). Given two setups and two applications, we randomized the order of the setups and the applications to reduce learning effects, so we had four groups. We randomly assigned the 12 participants to groups while maintaining even group sizes (three participants each).

[3]Evaluation replication package: https://bit.ly/unfold-evaluation.

[4]Modified from examples found in intermediate-level tutorials on JavaScript event handling: bit.ly/uf-memory-game and bit.ly/uf-calculator, respectively.

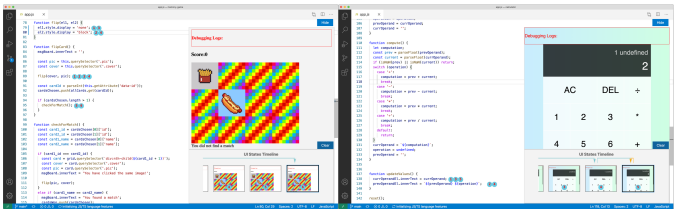

Fig. 4. The look of applications in our user study displayed in UNFOLD: Memory Game (left) and Calculator (right).

Procedure. Participants did the tasks via video conferencing. At the beginning of each task (application), we gave a 10-minute tutorial on the debugging setup using the same tutorial code and allowed them to ask clarifying questions. We told them two expected behaviors of the respective application (that correspond to the bugs listed in "Tasks") but not whether they were buggy. They then had 30 minutes to locate and fix the two bugs in their desired order. They were allowed to look up API documentation, encouraged to think aloud, and required to tell us when they located or fixed a bug. We stopped the timer for the application when they either asked to move on or used up all 30 minutes. After debugging two applications, we spent the final 15 minutes giving them a survey and a semi-structured interview.

Quantitative Data. We recorded screencast videos of each session and measured (1) duration and (2) success of both locating and fixing bugs by reviewing the recordings. We measured the time to locate a bug as the duration from the start of working on an app or the end of locating/fixing the previous bug until the participant said they found it and objectively did so based on the experimenter's judgment, or the time was up. Specifically, we counted the timestamp when participants located a bug by them correctly identifying the line(s) that caused it and verbally informing experimenter of correct location and cause. Participants who failed to locate any bug were marked as having taken 30 minutes (total time allowed for an application). Similarly, we measured the time of fixing a bug as the duration from the end of locating the bug, if any, until the participant said they fixed it or time was up. Finally, we marked that a participant succeeded in locating bugs in one app if both bugs were correctly located, and that they succeeded in fixing bugs if both bugs were correctly located and fixed.

Qualitative Data. We captured participants' debugging behavior via video recordings of the study and our notes. We recorded participant quotes during the study and interview. We also obtained open-ended responses and Likert-scale ratings on features of UNFOLD from the survey we gave right after the tasks were done. One author assembled the behavioral findings

into groups using category analysis [48], and coded the quotes and responses using thematic analysis [49].

Study Limitations. First, although we recruited participants from a variety of backgrounds with a broad range of programming experience, there were only 12 participants, who might not be representative enough, and whose self-reported programming experience might have biased the study results. Second, there were only two small GUI apps used in our study, and we only compared UNFOLD to one traditional debugging approach (Firefox DevTools). Thus, we do not know how well this technique generalizes to larger, more complex web applications that run in production environments. As such, our study findings should be viewed as an early step towards evaluating the effects of live programming for GUI applications or implementations of future live programming tools for this domain.

VI. RESULTS

A. Quantative Results: Effectiveness on Debugging

To investigate how live programming affects debugging GUI applications, we compared the success and duration of (1) locating bugs and (2) fixing bugs across both settings in both tasks. We report these quantitative results below.

While two participants failed to locate bugs in the CONTROL condition, only one failed with UNFOLD. In Memory Game, five out of six participants located all bugs in UNFOLD and six did so in the CONTROL condition. In Calculator, six participants located all bugs in UNFOLD and four out of six did so in CONTROL. The differences in results are not statistically significant according to Fisher's exact test. However, participants who failed to locate bugs in each respective condition presented different behavior. When using CONTROL, P9 and P12 who failed to locate bugs indeed spent all the task time on localization. In contrary, P8 who failed to locate bugs within task time when using UNFOLD spent half of the task time just annotating the code line by line, doing little relevant to finding bugs, and only starting the localization task when prompted by the experimenter.

Participants located bugs significantly faster with UNFOLD in Calculator. Fig. 5 shows the duration of locating bugs by application and debugging setup. Since participants were given 30 minutes to work on each application, including locating and fixing bugs, the maximum time one could spend locating bugs was 30 minutes. Using a Wilcoxon signed-rank test on median values, we found that participants with UNFOLD located bugs *significantly* faster in Calculator, by 15.7 minutes ($p = .036$), but only marginally faster in Memory Game ($p = .562$), by 0.9 minutes.

Regardless of the condition, participants struggled to fix bugs. We did not observe any difference in the success or duration of fixing bugs across the conditions. In Memory Game, three out of six participants fixed all bugs in UNFOLD, and three out of six did so in the CONTROL condition. In Calculator, three out of six participants fixed all bugs in UNFOLD, and two out of six did so in CONTROL. Fisher's exact test did not show any significant difference caused by the debugging setup in the success of fixing all bugs. Comparing the duration of those who eventually succeeded in fixing the bugs, we found through median values that participants using UNFOLD spent 3.5 minutes more fixing bugs in Memory Game but 3 minutes less in Calculator than those with CONTROL. The sample sizes were too small (all ≤ 3) to conduct statistical tests and conclude any effects.

B. Qualitative Results: Debugging Strategies

Through video recordings and study notes, we derived the following qualitative insights on participants' debugging strategies affected by the tool condition.

When using UNFOLD, participants relied on runtime information more during debugging. With UNFOLD, participants used one event sequence as the input for liveness to get runtime information and see how their edits continuously affected the runtime execution. For example, P5 inspected the UI States Timeline to find out intermediate UI states that diverged from their expectations, used the UI-to-Code Connector Labels to locate the corresponding code in the editor, edited the code, and checked the updated UI States Timeline to confirm their hypothesis about the bug. Liveness further helped P1 and P7 realize some code they had added was unnecessary and wrong, with P7 commented, *"the [UI-to-code connector labels] indicate that the code I added is problematic... I immediately see the problem."* In contrast, although participants also interacted with the application to test their changes during debugging when in CONTROL, they focused more on reading and reasoning about code than interacting with the application. Notably, P3 successfully located and fixed all the bugs when in CONTROL, but did not do so via using any console logs, debugger, or web inspector at all, entirely by reading code and reasoning. This might be due to the fact that the cost of incremental testing is relatively high in our CONTROL implementation: it requires hard-reloading the application whenever the code changes through a combination of keyboard shortcuts. Such observations, although not surprising, show that our implementation of liveness in UNFOLD is reliable enough for programmers to continuously obtain runtime information.

When using UNFOLD, participants lacked mechanisms for deeper inspection of the visual state. We observed that, besides the understanding of the runtime behavior, deeper inspection of an app's visual state is also critical for resolving its bugs. In the study, participants could use the web inspector for directly inspecting the visual state when in CONTROL, but they had to log such information to the UI states timeline when using UNFOLD, which might not work well for deeper inspection. Particularly, the Memory Game task had one bug in setting CSS properties. P9 and P12, who located the bug using UNFOLD but failed to fix it, commented that the issue was not easily discoverable even with runtime information and wished they could directly inspect the visual state. P6 used logs in UNFOLD heavily for getting information about visual properties, causing a clutter in the UI states timeline and information overload. Meanwhile, participants when in CONTROL

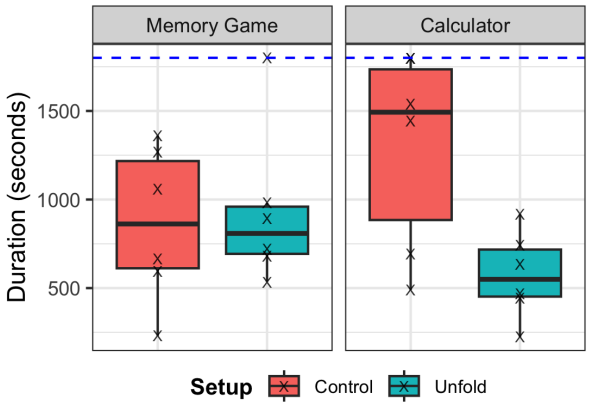

Fig. 5. Duration of locating bugs by application and setup. The "X" marks represent raw data points. The only overlapping data points (two) are at the upper left corner of the plot for Calculator (right). The center lines show the median values. The dashed horizontal line represents the time limit to one application (1800 seconds = 30 minutes).

could easily use the web inspector for examining/adjusting details of the DOM and CSS properties. For the same bug in Memory Game, P5, P6, and P11 used the web inspector when in CONTROL and found it quickly.

In addition to the observations about using UNFOLD, we also had findings general to live programming within the context of debugging GUI applications.

Even with liveness, participants started comprehension with a significant amount of code reading. In both conditions, when first starting the task with unfamiliar code, all participants read the static code for comprehension without interacting with the application. However, when using UNFOLD, half of the participants (P1, P2, P4, P7, P10, P11) would interact with the application briefly after reading the code to obtain information about the execution. P7 interacted with the application immediately after skimming the code and deemed the live information *"very straightforward"* for confirming their understanding. P11 further used UI-to-Code Connector Labels and logs to confirm parts of the execution and narrow down the search for bugs. In contrast, in CONTROL, almost all participants avoided interactions to the application when familiarizing themselves with the code, relying on reading static code only.

Full liveness is unnecessary for testing when programmers focus on final state only. For participants who were able to fix bugs, when they moved onto testing their edits more thoroughly, they used more than one event sequences. With UNFOLD, participants (P1, P2, P3, P7, P10) always cleared existing event inputs and provided new ones to the debugger, focusing on only the final state but not the intermediate UI states caused by the events. The unnecessary intermediate states in fact caused UNFOLD to slow down and led to complaints (P4, P5). In CONTROL, similarly, participants always reloaded the whole application to test the program behavior in full with different event input sequences.

TABLE I
LIKERT-SCALE USER PERCEPTIONS OF UNFOLD FEATURES FROM 1 - "STRONGLY DISAGREE" TO 5 - "STRONGLY AGREE". "AVG." - AVERAGE, "MDN." - MEDIAN, "DIST." - DISTRIBUTION.

	Ease of use			Helpfulness		
Feature	Avg.	Mdn.	Dist.	Avg.	Mdn.	Dist.
Event Recording	4.67	5.0		4.83	5.0	
Event Arrows	4.67	5.0		4.25	4.5	
Connector Labels	4.25	4.5		4.33	4.0	
Logging	4.08	4.0		4.17	5.0	
Live Feedback	4.00	4.0		4.42	5.0	
Overall	4.42	5.0		4.50	4.5	

C. Users' Perceptions of UNFOLD

Participants felt that UNFOLD reduces barriers to debugging GUI applications. The left part of Table I shows the ratings on how easy it was to use UNFOLD on a 5-point Likert scale. Participants generally found it easy to use: all ratings have averages and medians above 4 out of 5 (Strongly Agree). They mentioned three particular aspects of UNFOLD that ease debugging GUI apps.

Straightforward Interaction Flow. Participants found the UNFOLD interface to be *"very intuitive"* (P2, P3). P1 especially liked *"its simplicity,"* and P2 appreciated the flat learning curve.

Minimal Configuration Requirement. All participants found that UNFOLD was *"easy to setup,"* required *"[few] context shifts"* (P8), and picked up their *"existing [work practice] ... without [requiring] configuring what events to record"* (P2).

Minimal Distraction. Participants considered that features in UNFOLD did not distract them much from debugging. Although the UI-to-Code Connector Labels could be distracting *"when there [were] too many UI states"* (P3, P9, P11), the *"ease of resetting"* addressed this problem by clearing the recorded events and states.

Participants deemed UNFOLD helpful for debugging GUI applications. The right part of Table I shows participants' ratings for UNFOLD's helpfulness, also on a 5-point Likert scale. Broadly speaking, participants found it helpful, especially "Event Recording" that received an average rating of 4.83 out of 5. Participants deemed UNFOLD helpful in two specific ways.

UI States-to-Code Connections. 10 out of 12 participants mentioned that UNFOLD helped them reason about the behavior of the invoked event handlers by visually showing UI states and connecting the states to code execution. P8 especially appreciated the ability to *"quickly understand the effects of [their] actions."*

Live Feedback via Event Recording. Eight participants said UNFOLD greatly helped them debug by automating recorded events and providing live feedback upon code changes using

these events. P7 further commented that, when trying to understand the effects of edits, they were able to *"quickly compare the difference in app behavior between the changes"* through the live feedback.

Participants yearned for more control over liveness in UNFOLD. Participants also suggested possible improvements to UNFOLD that allow them to: (1) inspect runtime data and visual structure along with the UI states without logging (P2); (2) filter uninteresting UI states and connector labels (P3, P9, P10, P11); (3) examine UI components in detail in a UI state (P6); and (4) adjust the granularity of information shown (P11). The suggestions share a common theme: *more control over the granularity of liveness.*

VII. DISCUSSION AND FUTURE WORK

How does liveness help debugging? Our findings show that live programming helps programmers locate some GUI application bugs faster. We seek to understand this observation from our study, where the buggy code was not written by the participants themselves. We first refer to the *forward reasoning* theory [50], [51], which suggests that programmers tend to first simulate and reason about code execution in their head when locating bugs in unfamiliar code. Liveness precisely provides such help for simulating code execution by directly showing execution state changes to the programmer, which could be used to validate their mental model for the execution. Another possible explanation for the helpfulness of live programming in locating bugs is that it enables rapid edit-run cycles, which prior work suggests are frequent in debugging [17]. Indeed, we observed that participants relied on runtime information more during debugging when using UNFOLD. Live programming supports rapid edit-run cycles by automating program execution and continuously visualizing program behavior in full.

We do see that live programming helps with locating bugs more in some tasks than in the others. For example, one bug in the Memory Game task was better resolved with direct inspection of the visual structure than reasoning about the runtime execution. This possibly explains why we observed a significant speedup in locating bugs caused by live programming only in Calculator but not in Memory Game. *We thus conclude that live programming helps debugging in situations amenable to reasoning about runtime execution.*

Liveness is suitable for locating bugs, but more help is needed for fixing bugs. In our study, P9 and P12 located the bugs in Memory Game using UNFOLD but did not know how to fix them. There are two possible causes behind the failures: (1) they were unfamiliar with the API calls that caused the bugs, which could be addressed via searching documentation and code examples but require more time and context switches; (2) they had an expected UI state in mind but did not know how to achieve it via code. Indeed, while live programming may help understand code behavior and locate bugs, it cannot suggest fixes. We believe that *program synthesis* is well-suited for suggesting API usage and code edits. For example, millions are using the GitHub Copilot synthesizer [52] to obtain code suggestions within their IDE. Particularly for reaching expected UI states, synthesis via direct manipulation can also help and has been well-explored [53]–[55]. Still, an open problem remains in validating the behavior of synthesized code against the programmer's expectation. Recent work has observed the potential benefits of liveness for lowering the cognitive load of validating AI-generated code [16]. We believe that live programming, by showing correspondences between execution state changes and the code, can provide further help with understanding and refining the synthesized repair suggestions. *Future debuggers could help programmers not only locate but also fix bugs by integrating aspects of live programming and program synthesis.*

***To be live or not to be live?* More investigation needed**. Our study reveals two interesting yet contradictory limitations of live programming. First, live programming is sometimes *not live enough*: if the user does not provide program inputs for it to generate runtime information, they cannot benefit from live programming at all. Indeed, participants in our study, even when using UNFOLD, did not immediately use liveness for understanding unfamiliar code at the very beginning of a task, likely because interacting with the application (as program input) was not their first reaction. Second, live programming can be *too live*: if the user is not interested in intermediate runtime states and their connections to the code, such information can be too distracting. In our study, participants focused on final state only when systematically testing their bug fixes, and those using UNFOLD with live programming complained about the unnecessary intermediate states in this setting and wanted more control over liveness.

To alleviate these limitations, we believe different phases of the programming workflow require different levels of liveness. This connects with prior work [9] that different programmers opt for different kinds of liveness at different points. Customizable liveness is one way to address the limitations, as four participants in the post-study interview suggested having customizable display of intermediate program states. In addition to user-initiated customization, we further believe that live programming environments could *actively* show the appropriate levels of liveness at the right time. When the programmer just started reading unfamiliar code, which is common for debugging in collaborative settings, the environment could auto-generate program inputs to encourage using liveness for comprehension. When the programmer enters the final testing phase, liveness could be reduced to showing final states only upon different program inputs. The bigger challenge towards enabling the above is detecting which phase the programmer is in. *Future live programming environments could optimize their help and minimize distractions by detecting the programmer's workflow and self-adjusting the granularity of liveness.*

VIII. CONCLUSION

In this work we first explored the design of a live programming environment for debugging GUI applications through a needfinding study. Then following design goals derived from

this study, we proposed UNFOLD, a prototype live programming environment for debugging web-based GUI applications. We evaluated UNFOLD in a within-subjects study with 12 participants, and found that liveness helped programmers locate some kinds of GUI application bugs faster, encouraged programmers to use runtime information for debugging, and was deemed easy to use and helpful. Our findings can inform opportunities for future live programming and debugging tools in terms of adjustable liveness and support for fixing bugs.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments and suggestions. This work was supported in part by NSF grant 2107397.

REFERENCES

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug. 2016, conference Name: IEEE Transactions on Software Engineering.

[2] D. Spinellis, "Modern debugging: the art of finding a needle in a haystack," *Communications of the ACM*, vol. 61, no. 11, pp. 124–134, Oct. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3186278

[3] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia, "Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Oct. 2013, pp. 383–392, iSSN: 1949-3789.

[4] B. Myers, R. McDaniel, R. Miller, A. Ferrency, A. Faulring, B. Kyle, A. Mickish, A. Klimovitski, and P. Doane, "The Amulet environment: new models for effective user interface software development," *IEEE Transactions on Software Engineering*, vol. 23, no. 6, pp. 347–365, Jun. 1997, conference Name: IEEE Transactions on Software Engineering.

[5] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive record/replay for web application debugging," in *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*. St. Andrews Scotland, United Kingdom: ACM, Oct. 2013, pp. 473–484.

[6] B. Victor, "Learnable Programming," http://worrydream.com/LearnableProgramming/, 2012.

[7] S. L. Tanimoto, "A perspective on the evolution of live programming," in *2013 1st International Workshop on Live Programming (LIVE)*, May 2013, pp. 31–34.

[8] C. M. Hancock, "Real-time programming and the big ideas of computational literacy," Thesis, Massachusetts Institute of Technology, 2003. [Online]. Available: https://dspace.mit.edu/handle/1721.1/61549

[9] S. Lerner, "Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, ser. CHI '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3313831.3376494

[10] T. Lieber, J. R. Brandt, and R. C. Miller, "Addressing misconceptions about code with always-on programming visualizations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '14. New York, NY, USA: Association for Computing Machinery, Apr. 2014, pp. 2481–2490. [Online]. Available: https://doi.org/10.1145/2556288.2557409

[11] L. Cabrera, J. H. Maloney, and D. Weintrop, "Programs in the Palm of your Hand: How Live Programming Shapes Children's Interactions with Physical Computing Devices," in *Proceedings of the 18th ACM International Conference on Interaction Design and Children*. Boise, ID, USA: ACM, Jun. 2019, pp. 227–236.

[12] C. Zhao, I.-C. Shen, T. Fukusato, J. Kato, and T. Igarashi, "ODEN: Live Programming for Neural Network Architecture Editing," in *27th International Conference on Intelligent User Interfaces*, ser. IUI '22. New York, NY, USA: Association for Computing Machinery, Mar. 2022, pp. 392–404. [Online]. Available: https://doi.org/10.1145/3490099.3511120

[13] R. A. DeLine, "Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-specific Language," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Yokohama Japan: ACM, May 2021, pp. 1–11.

[14] R. DeLine and D. Fisher, "Supporting exploratory data analysis with live programming," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Atlanta, GA: IEEE, Oct. 2015, pp. 111–119.

[15] M. Campusano, A. Bergel, and J. Fabry, "Does live programming help program comprehension?–A user study with Live Robot Programming," in *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. Amsterdam, Netherlands: ACM, Nov. 2016, p. 8 pages. [Online]. Available: http://bergel.eu/MyPapers/Camp16-ComprehensionWithLRP.pdf

[16] K. Ferdowsi, R. L. Huang, M. B. James, N. Polikarpova, and S. Lerner, "Validating ai-generated code with live programming," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, ser. CHI '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3613904.3642495

[17] A. Alaboudi and T. D. LaToza, "Edit - Run Behavior in Programming and Debugging," in *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct. 2021, pp. 1–10.

[18] "Introduction to events - learn web development," Feb 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events

[19] S. L. Tanimoto, "VIVA: A visual language for image processing," *Journal of Visual Languages & Computing*, vol. 1, no. 2, pp. 127–139, Jun. 1990.

[20] D. Rauch, P. Rein, S. Ramson, J. Lincke, and R. Hirschfeld, "Babylonian-style Programming," *The Art, Science, and Engineering of Programming*, vol. 3, no. 3, pp. 9:1–9:39, Feb. 2019.

[21] C. Omar, I. Voysey, R. Chugh, and M. A. Hammer, "Live functional programming with typed holes," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–32, Jan. 2019.

[22] X. Zhang and P. J. Guo, "DS.js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. Québec City QC Canada: ACM, Oct. 2017, pp. 691–702.

[23] E. Senft, M. Hagenow, R. Radwin, M. Zinn, M. Gleicher, and B. Mutlu, "Situated Live Programming for Human-Robot Collaboration," in *The 34th Annual ACM Symposium on User Interface Software and Technology*. Virtual Event USA: ACM, Oct. 2021, pp. 613–625.

[24] R. L. Huang, K. Ferdowsi, A. Selvaraj, A. G. Soosai Raj, and S. Lerner, "Investigating the Impact of Using a Live Programming Environment in a CS1 Course," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2022. New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 495–501.

[25] J. Kramer, J. Kurz, T. Karrer, and J. Borchers, "How live coding affects developers' coding behavior," in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Jul. 2014, pp. 5–8, iSSN: 1943-6106.

[26] C. Schuster and C. Flanagan, "Live Programming for Event-Based Languages," in *Proceedings of the 2015 Reactive and Event-based Languages and Systems Workshop, REBLS*, vol. 15, 2015. [Online]. Available: https://users.soe.ucsc.edu/~cormac/papers/15rebls.pdf

[27] R. Dey, "VSCode Live Server," Sep. 2022. [Online]. Available: https://github.com/ritwickdey/vscode-live-server

[28] ——, "VSCode Live Server++," 2022. [Online]. Available: https://github.com/ritwickdey/vscode-live-server-plus-plus

[29] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato, "It's alive! continuous feedback in UI programming," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 95–104.

[30] "Hot Reload — Vue Loader," https://vue-loader.vuejs.org/guide/hot-reload.html#state-preservation-rules, 2022.

[31] "react-refresh," 2022. [Online]. Available: https://www.npmjs.com/package/react-refresh

[32] J. Lincke, P. Rein, S. Ramson, R. Hirschfeld, M. Taeumel, and T. Felgentreff, "Designing a live development experience for web-components," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience*. Vancouver BC Canada: ACM, Oct. 2017, pp. 28–35.

[33] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding JavaScript event-based interactions," in *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad India: ACM, May 2014, pp. 367–377.

[34] P.-Y. P. Chi, S.-P. Hu, and Y. Li, "Doppio: Tracking UI Flows and Code Changes for App Development," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. Montreal QC Canada: ACM, Apr. 2018, pp. 1–13.

[35] P. Jiang, F. Sun, and H. Xia, "Log-it: Supporting Programming with Interactive, Contextual, Structured, and Visual Logs," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Hamburg, Germany: ACM, New York, NY, USA, 2023, p. 16. [Online]. Available: https://doi.org/10.1145/3544548.3581403

[36] A. Alaboudi and T. D. Latoza, "Hypothesizer: A hypothesis-based debugger to find and test debugging hypotheses," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3586183.3606781

[37] S. Oney and B. Myers, "FireCrystal: Understanding interactive behaviors in dynamic web pages," in *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sep. 2009, pp. 105–108.

[38] B. Burg, A. J. Ko, and M. D. Ernst, "Explaining Visual Changes in Web Interfaces," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. Charlotte NC USA: ACM, Nov. 2015, pp. 259–268.

[39] J. Hibschman and H. Zhang, "Unravel: Rapid Web Application Reverse Engineering via Interaction Recording, Source Tracing, and Library Detection," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ser. UIST '15. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 270–279.

[40] ——, "Telescope: Fine-Tuned Discovery of Interactive Web UI Feature Implementation," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, ser. UIST '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 233–245.

[41] W. Wang, G. Fraser, M. Bobbadi, B. T. Tabarsi, T. Barnes, C. Martens, S. Jiao, and T. Price, "Pinpoint: A Record, Replay, and Extract System to Support Code Comprehension and Reuse," in *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Computer Society, Sep. 2022, pp. 1–10.

[42] H. Beyer and K. Holtzblatt, *Contextual Design: Defining Customer-Centered Systems*. San Francisco, Calif: Morgan Kaufmann, 1998.

[43] C. Developers, "Chrome DevTools - Overview," 2022. [Online]. Available: https://developer.chrome.com/docs/devtools/overview/

[44] "Firefox DevTools User Docs — Firefox Source Docs documentation," https://firefox-source-docs.mozilla.org/devtools-user/, 2022.

[45] A. Inc, "Tools - Safari," https://developer.apple.com/safari/tools/, 2022.

[46] "Visual studio code - code editing." 2023. [Online]. Available: https://code.visualstudio.com/

[47] J. Nielsen, "Iterative user-interface design," *Computer*, vol. 26, no. 11, pp. 32–41, Nov. 1993, conference Name: Computer.

[48] D. Yanow, "Qualitative-interpretive methods in policy research," in *Handbook of Public Policy Analysis*. Routledge, 2017, pp. 431–442.

[49] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, Jan. 2006.

[50] I. Katz and J. Anderson, "Debugging: An Analysis of Bug-Location Strategies," *Human-Computer Interaction*, vol. 3, no. 4, pp. 351–399, Dec. 1987. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1207/s15327051hci0304_2

[51] P. Romero, B. du Boulay, R. Cox, R. Lutz, and S. Bryant, "Debugging strategies and tactics in a multi-representation software environment," *International Journal of Human-Computer Studies*, vol. 65, no. 12, pp. 992–1009, Dec. 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1071581907001000

[52] "GitHub Copilot · Your AI pair programmer," https://github.com/features/copilot, 2022.

[53] C. Schuster and C. Flanagan, "Live programming by example: using direct manipulation for live program synthesis," in *LIVE Workshop*, 2016. [Online]. Available: https://chris-schuster.net/live16/live16-lpbe.pdf

[54] B. Hempel and R. Chugh, "Maniposynth: Bimodal Tangible Functional Programming," Tech. Rep., Jun. 2022.

[55] B. Hempel, J. Lubin, and R. Chugh, "Sketch-n-Sketch: Output-Directed Programming for SVG," in *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, Oct. 2019, pp. 281–292.