Towards Live Programming for Interactive GUI Applications

RUANQIANQIAN (LISA) HUANG, UC San Diego, USA

Interactive GUI applications are crucial to human-computer interaction. Despite existing tools and frameworks that aim to reduce such complexity, understanding and debugging these applications remains yet a nontrivial problem facing programmers. The emergence and rise of live programming, a paradigm that results in real-time traces of each step of code execution that are always up-to-date with the code, brings hope to tackling this problem. However, while live programming has been applied to many settings, interactive GUI applications is an area where live programming has been under-explored.

In this report, I review two lines of research: (1) tools for understanding/debugging interactive GUI applications and (2) live programming environments. Based on existing literature, I then discuss why live programming could be helpful for program comprehension and debugging in general, and why using live programming for the development of interactive GUI applications is potentially promising yet remains an open problem. Finally, I introduce recent research efforts at UC San Diego in live programming for interactive GUI applications and propose other future directions.

1 INTRODUCTION

Interactive GUI applications are ones in which the user interacts with some Graphical User Interface (GUI), and the application uses event handlers to react to these user-interactions. They are crucial to human-computer interaction: most user-facing applications, such as web applications, email clients, word processors, and paint programs, are included in this set of interactive applications. As computing technologies evolve, the functionality of modern interactive GUI applications increases, so does the complexity of their development. Despite the existence of tools and frameworks that aim to reduce such complexity (e.g., [Alimadadi et al. 2014] and [Ko and Myers 2009]), understanding and debugging interactive GUI applications remains a nontrivial problem facing programmers [Burg et al. 2013; Oney and Myers 2009].

The emergence and rise of live programming brings hope to tackling the problem. Live programming is a paradigm in which runtime values at each step of program execution in a given context is always provided to the programmer. Live programming has been applied to a variety of settings, including the programming of general-purpose languages [Biegel et al. 2015; Kramer et al. 2014; Lerner 2020; Niephaus et al. 2020; Omar et al. 2019; Rauch et al. 2019], robot programming [Cabrera et al. 2019; Campusano et al. 2016; Senft et al. 2021], and data science [DeLine and Fisher 2015; DeLine 2021; Zhang and Guo 2017]. Qualitative feedback suggests that live programming is effective in program comprehension and debugging in various settings, from classroom [Huang et al. 2022] to industry [DeLine et al. 2015], and from analyzing data [DeLine and Fisher 2015] to tweaking neural network architecture [Zhao et al. 2022].

To realize live programming for interactive GUI applications, per the definition of live programming, there are two requirements that need to be met: (1) the input, which comes from user-interactions, and (2) the presentation of runtime information at each step of execution, which includes not only runtime data but also GUI changes. While there has been prior work on specific aspects of live programming for interactive GUI applications [Burckhardt et al. 2013; Dey 2022a,b; Lincke et al. 2017; Schuster and Flanagan 2015; Source 2022; Vue.js 2022], the full potential for live programming in the setting of interactive GUI applications has yet to be fulfilled.

In this report, I review two lines of research: (1) tools for understanding/debugging interactive GUI applications and (2) live programming environments. I discuss the potential of using live programming for interactive GUI applications and open problems that need to be addressed. The rest of this report is structured as follows:

Author's address: Ruanqianqian (Lisa) Huang, r6huang@ucsd.edu, UC San Diego, La Jolla, CA, USA.

- Sec. 2 provides an overview of existing tools for understanding interactive GUI applications and tools for debugging, and discusses the design space of tools for both understanding and debugging interactive GUI applications implied from such tools.
- Sec. 3 reviews the history of live programming, the domains to which it has been applied, and evidence of its effectiveness.
- Based on these two lines of research as well as literature in cognitive science, Sec. 4 discusses why it is promising to apply live programming to understanding and debugging interactive GUI applications, summarizes existing work towards this direction, and identifies open problems yet to be addressed in existing work.
- Sec. 5 introduces LiveUP, recent work led by the report author in live programming for interactive GUI applications that aims to address the open problems.
- Finally, Sec. 6 proposes possible next-steps to further the research in live programming for interactive GUI applications.

2 TOOLS FOR INTERACTIVE GUI APPLICATIONS

While there are many tools addressing different aspects of the development of interactive GUI applications, this section reviews only two types of such tools: tools for understanding interactive GUI applications, and tools for debugging.

In general, program understanding (or *comprehension*) and *debugging* are among the most critical and inter-connected components of software development [Gould 1975]. The program comprehension process uses existing knowledge and new knowledge acquired through the existing knowledge "to build a mental model of the software that is under consideration." [Von Mayrhauser and Vans 1995] Debugging, on the other hand, is "an activity that comes after testing" in which programmers figure out "where the error is and how to fix it." [McCauley et al. 2008] Without understanding the program behavior that leads to the error in the first place, it is impossible to fix the error. As such, it becomes obvious that to debug successfully, one needs to first understand the program [Von Mayrhauser and Vans 1995].

In the context of developing interactive GUI applications, comprehension and debugging are even more important because such applications are complex by nature. Interactive GUI applications normally have an initial state for its user interface (UI), and the progression of UI states is programmed through event handlers (of events of various forms, such as mouse events and keyboard events) and function calls, which is the core of the dynamic behavior of these applications. Therefore, to successfully understand and, if necessary, debug an interactive GUI application, the programmer needs to form a correct mental model of the behavior of this application, i.e., a correct mental model for the UI state progression and the code to be executed.

In this section, I provide an extensive, not exhaustive, review of tools that aim to aid the comprehension (Sec. 2.1) and debugging (Sec. 2.2) of interactive GUI applications. I then extract the design space (Sec. 2.3) implied by these tools based on themes they share, which provides a foundational understanding of any tool that aims to assist the understanding and debugging of interactive GUI applications.

2.1 Tools for Understanding Interactive GUI Applications

Tools for understanding interactive GUI applications are mainly used to understand the behavior of interactive GUI applications with potential purposes of code adaptation and reuse. This subsection surveys some of them to understand the common themes among these tools. The majority of these tools are for understanding the behavior of web applications [Alimadadi et al. 2014; Burg et al. 2015; Hibschman and Zhang 2015, 2016; Oney and Myers 2009], while there are also tools for understanding interactive GUI applications written in Alice [Gross et al. 2010] and Snap! [Wang et al. 2022], as well as Android applications [Chi et al. 2018]. To gain insights from how users gain information about the behavior of the interactive GUI applications from these tools, I review them from two aspects: how users interact with these tools to obtain such information, and how users navigate themselves among the information for comprehension.

Obtaining Information for Comprehension. To see how an application responds to a user-interaction or a sequence of user-interactions, the interactions have to be provided first. However, in reality, the invocation of such interactions and the application's responses can be too quick to be inspected for comprehension, and sometimes the user needs to repeatedly invoke such interactions to reinforce their understanding of the associated response. As such, tools for understanding interactive GUI applications mostly adopts the technique of deterministic record/replay [Cornelis et al. 2003] so that the user only needs to provide the necessary user-interactions once, and can always replay the behavior of the application resulted from the recorded interactions. Some tools require *explicit* record/replay of user-interactions: the user explicitly presses a button in the tool's interface to start a recording of their user-interactions for later replaying and inspecting the UI changes and events caused by the recorded interactions [Gross et al. 2010; Hibschman and Zhang 2015; Oney and Myers 2009; Wang et al. 2022]. Other tools adopt *implicit* record/replay of the user-interactions: the user interacts with the GUI application as usual, such interactions automatically recorded, and the tool visualizes information it obtains about how the application responds to the recorded interactions [Alimadadi et al. 2014; Burg et al. 2015; Chi et al. 2018; Hibschman and Zhang 2016].

Navigating Information for Comprehension. Now that the user has obtained information about how an application reacts to their interactions, they need to use such information for help to truly understand the application's behavior, i.e., to build a correct mental model. A pattern I observe from the reviewed tools is that they all present information about the changes in the UI and the code executed that is responsible for such changes.

Some tools only present UI changes and the relevant code at the level of the entire UI [Chi et al. 2018; Gross et al. 2010; Oney and Myers 2009; Wang et al. 2022]. In other words, these tools capture any change to the entire UI and the code responsible for the change, and visualize the change-code records in the order of time. Other tools allow for focusing on UI changes and the relevant code at the level of individual UI components [Alimadadi et al. 2014; Burg et al. 2015; Hibschman and Zhang 2015, 2016], which can be useful when the user's only interest of observation is in a subset of UI components within the entire UI. It should be noted that, nevertheless, existing literature does not discuss which granularity of UI changes inspection is better, at the level of the entire UI or specific UI components.

In addition, some tools provide more control over the presentation of code relevant to the application's behavior. Some of them let the user focus on a specific subset of code execution [Alimadadi et al. 2014; Burg et al. 2015; Hibschman and Zhang 2016; Wang et al. 2022]. In particular, Telescope [Hibschman and Zhang 2016] allows for only showing the executed code that is responsible for UI changes and not from a third-party library. Pinpoint [Wang et al. 2022] allows the user to focus only on a subset of execution steps. Clematis [Alimadadi et al. 2014] shows code/information at three semantic levels. Scry [Burg et al. 2015] allows for side-by-side comparison of two UI states and explains the execution of HTML/CSS/JavaScript code behind the UI differences.

Summary. In general, the tools reviewed in this subsection share two main themes.

First, most tools for understanding interactive GUI applications use deterministic record/replay [Cornelis et al. 2003] to obtain information that facilitates understanding, so that users can continuously access such information by providing necessary user-interactions only once. Note that the recording of user-interactions could be either explicit or implicit. While existing work does not comment on which record/replay mechanism might be preferred, there are pros and cons in both mechanisms. Explicit record/replay can be ideal for explicitly prompting the users to express their intent to get the information for comprehension, but it comes with additional configuration overheads. Implicit record/replay can be preferred for not requiring such configuration overheads, but tools with implicit record/replay may present inspection information unnecessarily when the user does not want such information, causing information overload. All said, having *some* record/replay mechanism seems to be essential in tools for understanding interactive GUI applications.

Second, all of the tools reviewed above present visual mapping between UI changes and relevant code regardless of level level of visualization granularity, aiming to provide visual connection between code execution and interface changes. While it seems that more granular examination (e.g. at the level of individual UI components) would be preferred for applications with a complicated UI [Burg et al. 2015; Hibschman and Zhang 2016], all tools support inspecting UI changes at the level of the entire UI.

For the main purpose of this report, I am interested in the shared themes across these tools to understand the design space they imply. However, it is also important to understand the differences among these tools, including:

(1) The mechanism they adopt for the user to provide necessary interaction inputs in order to obtain information for comprehension, such as explicit versus implicit deterministic record/replay; and
(2) The different levels of granularity in presenting the information for user navigation, such as changes in the entire UI and in specific UI components.

2.2 Tools for Debugging Interactive GUI Applications

Now that I have reviewed a set of tools for understanding the behavior of interactive GUI applications, a question might arise as follows: *If these comprehension tools are already useful, why do people not use these tools for finding and fixing bugs in GUI applications, but rather develop a separate category of tools for debugging interactive GUI applications?* While debugging tools for interactive GUI applications have some overlap with tools for comprehension in their ability to reveal software misbehavior through some indicators, debugging tools distinguish themselves from comprehension tools by supporting editing and executing the applications from source [Alaboudi and LaToza 2021]. Moreover, it is important to have debugging tools specifically designed for interactive GUI applications as opposed to using generic debuggers because the applications' "highly dynamic, event-driven programming model stymies traditional static program analysis techniques." [Burg et al. 2013] This subsection reviews debugging tools for interactive GUI applications from the following aspects: the setting in which they are used, what role they play in the debugging process, and their support for the edit-run cycles [Alaboudi and LaToza 2021] in debugging.

Accessing the Tools for Debugging. The debugging tools reviewed in this report include standalone software [Ko and Myers 2009; Myers et al. 1996], a debugger built-into an existing IDE [Barr et al. 2016], tools with their own IDEs as well as APIs that can be incorporated into the code of an application [BDD 2022; Cyp 2022; Sel 2022], and installable or native extensions to a browser [Apple Inc. 2022; Burg et al. 2013; Chrome Developers 2022; Mozilla 2022].

Using the Tools for Debugging. Some debugging tools require users to edit the code for the application in some separate editor, then run the application and inspect its behavior via additional keyboard- or mouse-based configuration inside the tools' interfaces. UI testing tools like Selenium and similar alternatives [BDD 2022; Cyp 2022; Sel 2022] allow the user to write and edit test cases, run the tests, and inspect software misbehavior locally all in the tools' own interfaces, but the user has to use an external editor that does not come with such tools to perform source code edits. Timelapse [Burg et al. 2013] is used as a browser debugger where the user can record/replay their interactions, set breakpoints within the execution, and examine the runtime information. However, Timelapse also requires the user to edit the code for the application in an external editor and reload the application in the browser for the edits to take effect, although the previously recorded interactions and breakpoints will be kept. Similarly, existing browser developer tools, including the debugger and the web inspector, allow for changing the application code in the same interfaces and showing the immediate effect of the changes. Still, they require the user to manually migrate their temporary changes to the source code [Apple Inc. 2022; Chrome Developers 2022; Mozilla 2022]. Such limitation also leads to the limited support of re-execution through app reloading by browser developer tools: whenever the application is reloaded, all the temporary code changes made within the browser tools will be gone.

Some debugging tools support code editing and runtime examination all in one place [Barr et al. 2016; Ko and Myers 2009; Myers et al. 1996]. In particular, Amulet lets the user edit runtime value in the inspector window, but to apply such value changes to the source code, the user has to do so in Amulet's source code editor and reload the application to see the effect of their changes.

Finally, regardless of where the code editing and runtime examination should occur, all of the tools allow for in-depth examination of the running application: the user can pause in one particular execution step, and inspect the runtime information, the current code execution, and the current UI.

Support for Edit-Run Cycles. All of the debugging tools I have reviewed so far require a reload of the working application to see the effect of code edits (and whether the bugs have been fixed). The user will need to provide necessary user-interactions to trigger the behavior-to-debug as well, except in the UI testing tools [BDD 2022; Cyp 2022; Sel 2022] and Timelapse [Burg et al. 2013], which provide support for recording/replaying user-interactions.

Summary. The tools for debugging interactive GUI applications reviewed in this subsection share three main themes.

First, the reviewed debugging tools all allow for pausing at one particular state of the application's behavior progression and performing in-depth examination about its runtime state.

Second, all of the tools require various degrees of context switch between editing code and examining the runtime state of the application. Such context switch overheads include keeping the code editing and runtime examination in separate interfaces (sometimes applications), lacking support for user-interaction record/replay, and requiring a full reload of the application to examine the effect of code edits.

Third, and most importantly, the visual connection between code execution and interface changes that is commonly seen in comprehension tools (Sec. 2.1) is not as directly presented in the debugging tools. It is true that the debugging tools support in-depth examination of the application's behavior by supplying an affluent amount of runtime information at each step of the execution, including the runtime data, the call stack and the UI state that the execution produces. Still, comparing to the level of comprehension support widely available in the comprehension tools, it is worrisome that the support in the debugging tools for comprehension (which is necessary for debugging) is rather limited.

Granted that this report focuses more on the shared themes across these debugging tools to understand the design space they imply, it is also important to note the differences among these tools, including:

(1) How these tools are accessed, such as via existing browsers/IDEs or through standalone installations; and
(2) Where code editing and runtime inspection take place, such as in separate panes of one environment and in separate environments.

2.3 A Tool for Understanding and Debugging Interactive GUI Applications

In summary, Sec. 2.1 reviews some existing tools for understanding interactive GUI applications, in which I identified two themes in these tools:

(1) The use of deterministic record/replay to liberate users from manually, repeatedly providing necessary user-interactions; and
(2) The presentation of the visual mapping between UI changes and code responsible for the changes.

Sec. 2.2 reviews some tools for debugging interactive GUI applications, which share three themes:

(1) The ability to examine runtime behavior in-depth;
(2) The overheads of context switch between editing code and examining runtime behavior; and
(3) The limited support for comprehension, specifically the visual connection between code execution and interface changes (not as directly presented in the debugging tools while commonly seen in the comprehension tools).

As is discussed at the beginning of this section, a tool for debugging interactive GUI applications should provide support for comprehension as well, because to debug successfully one needs to understand the program first. However, while there is a whole category of tools that makes the understanding of interactive GUI applications intuitive (Sec. 2.1), such support for comprehension is rather limited in the debugging tools (Sec. 2.2).

Combining the shared themes for both tools for understanding interactive GUI applications and tools for debugging, I derive a design space of tools for both understanding *and* debugging interactive GUI applications that addresses three requirements, namely:

- **Ease of Inspection**: Automating the display of program execution via interaction record/replay;
- **Aid for Comprehension**: Visually connecting UI changes with relevant executed code; and
- **Support for Examining and Fine-Tuning Runtime Behavior**: Enabling in-depth examination of program (mis)behavior at runtime and quickly fine-tuning such (mis)behavior.

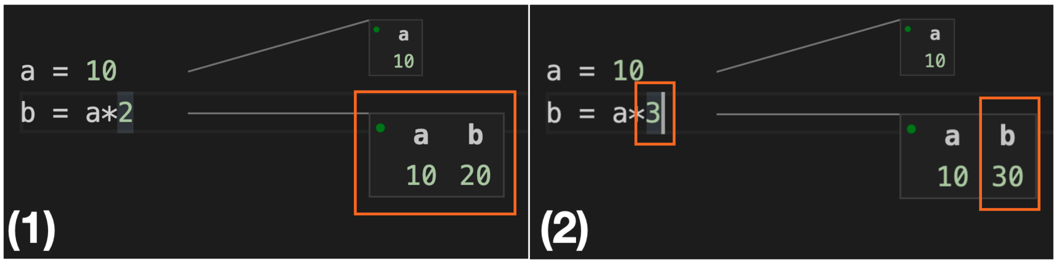
Note that most of the comprehension tools reviewed in Sec. 2.1 satisfy all requirements but “fine-tuning runtime behavior”. On the other hand, none of the debugging tools reviewed in Sec. 2.2 fully meet all of the three requirements: they either ask the user to repeatedly provide user-interaction inputs, or do not provide extensive aid for comprehension, or require additional overheads for modifying the program behavior and quickly seeing the effect of the changes. I will revisit these requirements in Sec. 4 when discussing why live programming environments Sec. 3 can potentially satisfy all three requirements for debugging and understanding interactive GUI applications, addressing this gap in literature.

3 LIVE PROGRAMMING

In this section, I review existing work in live programming. I survey live programming environments and the domains to which they each apply. I discuss two main kinds of liveness preserved in these tools, namely full-history liveness and single-state liveness. Finally, I summarize the qualitative and quantitative evidence from the literature that illustrates the potential effectiveness of live programming.

3.1 Definition and History

Live programming is a paradigm in which, given a context, immediate feedback on program execution is always provided to the programmer. Fig. 1 illustrates modifying a program in a live programming environment with Projection Boxes [Lerner 2020]. In Fig. 1-(1), the runtime values of variables `a` and `b`, respectively 10 and 20, are displayed in a box to the right. The box is called a *projection box*. In this program, variable `b` is declared to be the twice of the value of `a`. If we change



```
a = 10
b = a*2
```

```
a = 10
b = a*3
```

Fig. 1. Live programming with Projection Boxes [Lerner 2020]: (1) runtime values of initial program displayed in the projection boxes at the right; (2) runtime values of the program in the projection boxes immediately updated after the change in the second line.

the declaration of b to be three times of a, as in Fig. 1-(2), we see that the runtime value of b is immediately updated to be 30 in the projection box.

The notion of live programming seems to have been first studied in the context of visual languages [Tanimoto 1990]. Hancock's seminal work [Hancock 2003] lays the foundation of using live programming to promote the acquisition of computational thinking: If programming is modeling, then live real-time programming is live real-time modeling, and liveness bridges our mental, logical reasoning ("theories") to its impact on the physical world in which we are situated ("bodies"). Bret Victor's essay [Victor 2012] also calls for making programming learnable through liveness such that the programmer is able to follow along the program execution through visible states, which confirms with Hancock's argument on how liveness creates theories-bodies connection. Most recently, Tanimoto's 2013 essay [Tanimoto 2013] summarizes the main motivation for liveness: minimal latency, control over real-time effect , and support for learning.

3.2 Applications

Up until today, we have seen live programming environments applied to various domains. For general-purpose programming, live programming environments have been built around languages such as JavaScript [Kramer et al. 2014; Lieber et al. 2014; Rauch et al. 2019], Java [Biegel et al. 2015], Python [Kang and Guo 2017; Lerner 2020], an ML-like lanugage [Omar et al. 2019], and a Python-/JavaScript-like language [Kasibatla 2018]. It has also been recently prototyped in polyglot programming, in which a generic design of liveness is realized in multiple programming languages through the language server protocol [Niephaus et al. 2020]. Live programming is also seen in more specific domains including data science, [DeLine et al. 2015; DeLine and Fisher 2015; DeLine 2021; Zhang and Guo 2017], physical and robotic computing [Cabrera et al. 2019; Campusano et al. 2016; Senft et al. 2021], machine learning [Zhao et al. 2022], and direct manipulation systems [Wilcox et al. 1997]. In addition, there are a few live programming environments targeting interactive GUI applications, which will be reviewed in more detail when I discuss live programming for interactive GUI applications in Sec. 4.

Live programming environments are more than arbitrary research artifacts. In fact, many of them have been evaluated with target audience to obtain preliminary evidence about the effectiveness of live programming. There are also beginner-oriented live programming tools that have been evaluated with novices of the target domains to examine how live programming affects their early experiences in these domains [Cabrera et al. 2019; Campusano et al. 2016; Kang and Guo 2017; Senft et al. 2021; Wilcox et al. 1997; Zhang and Guo 2017; Zhao et al. 2022]; live programming environments targeting programmers with higher levels of experience have been evaluated similarly [Biegel et al. 2015; DeLine and Fisher 2015; DeLine 2021; Kramer et al. 2014; Lerner 2020]. Other than exploratory lab studies, live programming environments have also been used in real-world settings including classroom [Hancock 2003; Huang et al. 2022] and industry [DeLine et al. 2015], and distributed as extensible open-source software [Lerner 2020; Lieber et al. 2014; Niephaus et al. 2020]. Although there are several live programming environments that have not been evaluated with users [Kasibatla 2018; Niephaus et al. 2020; Omar et al. 2019; Rauch et al. 2019], their potential usability has been demonstrated through formalism or case studies.

3.3 Granularity of Liveness

Although all the tools reviewed in this section are considered as live programming environments, they present *liveness* at different levels of granularity. There are two main levels of granularity of liveness as seen in these tools: *full-history* and *single-state.* A few environments also provide the flexibility of switching between these two levels of granularity.

Environments with full-history liveness visualize how the runtime state changes throughout the execution [Kang and Guo 2017; Kramer et al. 2014; Lieber et al. 2014; Niephaus et al. 2020; Rauch et al. 2019; Wilcox et al. 1997; Zhao et al. 2022]. For example, Kramer et al. [Kramer et al. 2014] developed a prototype that continuously shows how JavaScript code executes step by step. Theseus [Lieber et al. 2014] implements full-history liveness for JavaScript, too, but it emphasizes more on the runtime control flow and function call counts as opposed to runtime data changes. Omnicode [Kang and Guo 2017] pushes full-history liveness to an extreme by showing the entire history of all runtime values for all variables at all time. Note that the environments above require an explicit invocation of a program to enable full-history liveness. As an alternative, Rauch et al. [Rauch et al. 2019] and Niephaus et al. [Niephaus et al. 2020] developed environments with full-history liveness that accepts program invocation through in-line examples (via UI widgets or test comments), which are inspired by the notion of example-based programming.

On the other hand, environments with single-state liveness provides immediate visual update on one particular state, often the final state of the program execution, as opposed to the full timeline of state changes overtime [Biegel et al. 2015; Cabrera et al. 2019; Campusano et al. 2016; DeLine et al. 2015; DeLine and Fisher 2015; DeLine 2021; Omar et al. 2019; Senft et al. 2021; Zhang and Guo 2017]. Note that the nature of some of the paradigms targeted by these environments determines the fact that they might benefit more from a design with single-state liveness. For example, in the context of data science [DeLine et al. 2015; DeLine and Fisher 2015; DeLine 2021; Zhang and Guo 2017], users are generally data scientists or end-users who only concern the final output of data processing and analysis, so it makes more sense to only provide feedback on the final state of the program. The other interesting domain would be physical computing [Cabrera et al. 2019; Senft et al. 2021], in which it is impossible to present/playback the full history of state changes in a live manner, because such changes occur in the real physical world.

Last but not least, some environments allow users to alternate between full-history liveness and single-state liveness depending on their preferences and task context [Kasibatla 2018; Lerner 2020]. Such customizability is actually positively perceived by users, as Lerner noted [Lerner 2020].

3.4 Effectiveness

Qualitative Evidence. Existing literature shows rich evidence on how positively users perceive of live programming. First, users find the live programming paradigm to be novice-friendly and intuitive to learn [Senft et al. 2021; Zhao et al. 2022]. Second, users consider live programming to be helpful in the following aspects:

Understanding program behavior [Biegel et al. 2015; Campusano et al. 2016; DeLine et al. 2015; Huang et al. 2022; Kang and Guo 2017; Kramer et al. 2014], through the immediate feedback provided by live programming that helps the user stay close to the program output [DeLine and Fisher 2015; DeLine 2021; Zhang and Guo 2017], as well as the full execution traces in environments with full-history liveness;

Debugging, especially noticing or locating bugs, specifically in environments with full-history liveness [Huang et al. 2022; Kang and Guo 2017; Lieber et al. 2014; Wilcox et al. 1997], in which sequential state changes become transparent and easy for inspection.

Although liveness can be distracting sometimes [DeLine et al. 2015; Huang et al. 2022] due to the amount of information it presents, users of Projection Boxes [Lerner 2020] suggest that such distraction can be alleviated when the liveness is configurable.

Quantitative Evidence. Apart from liveness itself, researchers suggest through quantitative evidence that the immediate feedback provided by live programming can benefit the short, frequent edit-run cycles in debugging [Alaboudi and LaToza 2021]. Researchers have also been directly measuring the effectiveness of live programming quantitatively through the user evaluations of several tools. For example, Cabrera et al. found that liveness seems to encourage users to interact more often and more extensively when programming physical devices [Cabrera et al. 2019]. Zhao et al. also observed promising quantitative results of live programming when running a user study on ODEN, in which they found that ODEN helped programmers of neural networks solve significantly more tensor shape mismatch errors. However, there are also results suggesting that live programming may not be as effective as we believe. While positive user perceptions of live programming were found in both a controlled lab experiment [Campusano et al. 2016] and an educational setting [Huang et al. 2022], researchers in either study noticed no significant improvement on the correctness or the time taken for program comprehension, or knowledge gain.

Summary. Is live programming effective? Although there has not been conclusive evidence, the results from user evaluations presented in existing literature, especially the qualitative results, shed some light on the effectiveness of live programming. Meanwhile, the fact that the paradigm is adaptable to various domains and settings is promising in that we might be able to obtain generalizable knowledge about the effectiveness of live programming across difference application domains and settings.

4 TOWARDS LIVE PROGRAMMING FOR INTERACTIVE GUI APPLICATIONS

Now that I have reviewed the design space implied by tools for understanding and debugging interactive GUI applications and live programming environments, I discuss in this section why live programming for interactive GUI applications is a promising direction and what needs to be addressed. I first explain why live programming can help program comprehension and debugging in general, and why a tool with live programming fits well within the space of tools for interactive GUI applications. I then summarize existing live programming artifacts for interactive GUI applications, and discuss requirements unfulfilled in these artifacts for live programming towards this direction.

4.1 Liveness for Program Comprehension and Debugging

When describing how humans interact with artifacts, Hutchins et al. [Hutchins et al. 1985] proposed the notions of the “gulf of execution” and the “gulf of evaluation”, which are later popularized by Norman’s book The Design of Everyday Things (originally published as The Psychology of Everyday Things [Norman 1988]). The *gulf of execution* refers to the gap between the user’s intentions about using a system and the system’s actual capabilities. The gulf of execution is small when the user’s intentions match well with the system’s capabilities. The *gulf of evaluation* refers to the gap between the system’s presentations of its capabilities and the user’s perceptions of the presentations. The gulf of evaluation is minimized when the user perceives sufficient information about the system’s behavior to build an accurate mental model for it. In other words, an accurate comprehension of the program leads to a minimal gulf of evaluation. **I argue that the live programming helps reduce the gulf of evaluation in programming in general.**

First, live programming allows the user to see the output of a program with given inputs, and to immediately evaluate how the output differs given the same inputs when a change is made to the program. Such immediate feedback helps the user to stay close to the program output when actively editing source code [DeLine 2021; Rauch et al. 2019], quickly refining their perceptions of the information implied by the program's execution. Empirical evidence also shows that liveness can help programmers quickly confirm and correct their mental models (their expectations) of program execution [Biegel et al. 2015; Campusano et al. 2016; Huang et al. 2022; Kang and Guo 2017]. Hence, live programming can help minimize the gulf of evaluation through its immediate feedback.

Second, existing literature shows that liveness can encourage more user interactions with the system to facilitate the user's understanding of the system's capabilities [Cabrera et al. 2019]. In programming, such interactions include program edits, inspections and (re-)executions. In cognitive science literature, Kirsh and Maglio argued that such interactions, defined as *epistemic actions*, help people understand the world they are in [Kirsh and Maglio 1994]. As such, in the context of programming, these *epistemic actions* help the programmer fine-tune their expectations of a program's behavior. It can be thus derived that live programming encourages more interactions with the code to help the user better understand the program, which eventually leads to a reduced gulf of evaluation.

In general, live programming helps reduce the gulf of evaluation in programming, which corresponds to a better understanding of the program that further helps debugging, as is discussed in Sec. 2. In addition, live programming directly facilitates debugging in the following aspects: (1) it eases locating erroneous code [DeLine and Fisher 2015; Huang et al. 2022; Kang and Guo 2017; Kramer et al. 2014; Lieber et al. 2014; Wilcox et al. 1997; Zhao et al. 2022], and (2) supports quick and frequent edit-run cycles in debugging [Alaboudi and LaToza 2021].

4.2 A Comprehension and Debugging Aid for Interactive GUI Applications

Based on the review of existing tools for understanding and debugging interactive GUI applications in Sec. 2, I consider live programming environments to be complying with the design space implied by these tools (Sec. 2.3), which makes them a good candidate for an aid for both comprehending and effectively debugging (which depends on comprehension) interactive GUI applications.

Automating the Visualization of Program Execution. Most comprehension tools reviewed in Sec. 2.1 for interactive GUI applications use record/replay techniques to display program execution information and explain visual changes without requiring the user to repeatedly provide user-interaction inputs. The comprehension tools, however, do not support editing the source code of an interactive GUI application and seeing how the code changes affect its behavior. The debugging tools reviewed in Sec. 2.2 allow the user to perform source code editing, but require additional overheads for examining the effect of code edits. Live programming constantly visualizes (and replays) the runtime execution of a program without requiring a restart of the program even when modified, addressing the limitations in both the comprehension tools and the debugging tools. Therefore, live programming seems to satisfy the requirement of tools for both understanding and debugging interactive GUI applications: automating the creation and update of visualizing program execution without repeatedly asking for inputs for the execution.

Visually Connecting Code to Output. All comprehension tools for interactive GUI applications reviewed in Sec. 2.1 visualize UI changes and the executed code responsible for the changes in order to explain (or, at least, help the user understand) how the code execution maps to the UI changes.

The tools for debugging interactive GUI applications provide some support for comprehension because comprehension is critical for debugging, but the support is not as comprehensive as in the comprehension tools (Sec. 2.2). Live programming environments with full-history liveness show the program output, let it be runtime values or control flows, of each step of code execution, and provide continuous updates whenever the code is modified. They can thus be used as debugging tools with extensive support for program comprehension (as is discussed in Sec. 3.4). As such, live programming environments, specifically environments with full-history liveness, satisfy the requirement of tools for both understanding and debugging interactive GUI applications by visually connecting each step of code execution to the UI change(s) the application produced.

Supporting the Examination and Fine-Tuning of Runtime Behavior. Both the tools for understanding interactive GUI applications and those for debugging provide some mechanism for helping the user examine program behavior at runtime to realize and locate misbehavior. Through either manual inspection and observation or noticeable visual cues, the user can locate code responsible for unexpected behavior in interactive GUI applications. In the context of live programming, environments with full-history liveness provide similar (and even more) aids for examining runtime behavior. In these environments, because each step of execution is visualized, when the code misbehaves or produces an error, the user can immediately notice the issue by looking at the runtime information presented by live programming. Live programming can thus assist with revealing program misbehavior in a similar way to existing tools for understanding and debugging interactive GUI applications.

However, as is discussed above, tools for understanding interactive GUI applications do not support source code editing, and thus it is impossible to fine-tune the runtime behavior of an application with such tools. On the other hand, users can edit source code and reload the application to see how the code changes populate to the runtime behavior using tools for debugging interactive GUI applications, but additional overheads are required (such as reloading and repeating the necessary user-interactions), which prevent users from quickly seeing the effect of their changes. Live programming at all levels of granularity allows for immediately seeing the effect of code changes on the final runtime state with low overheads, and full-history liveness further shows the progression of runtime states at each point of execution.

To sum up, live programming fulfills the requirement of supporting the examination and fine-tuning of runtime behavior for tools for debugging (and understanding) interactive GUI applications, and such immediacy of live programming further enables quick edit-run cycles in debugging [Alaboudi and LaToza 2021].

4.3 Requirements of Live Programming for Interactive GUI Applications

While the previous subsection discusses why live programming for interactive GUI applications can be promising, it also implies several requirements to be met in this context.

Obtaining the Input. By its definition, live programming continuously visualizes the execution of a program *in a given context*, usually in the format of inputs given to the program. For interactive GUI applications, such *contexts* or *inputs* are usually in the form of user-specified interactions with the applications, when we are concerned about how the UI state changes over time as a result of user-interactions. Applications without a GUI can accept inputs easily in a textual format, usually in a programmable way that requires few configuration overheads. However, a challenge remains in how to provide user-interactions to GUI applications in a similar way to enable live programming. Learning from existing tools for understanding interactive GUI applications, deterministic record/replay might be one way to go.

Visualizing the Execution Traces. Existing applications of live programming present real-time traces of code execution on the level of runtime data. In the context of interactive GUI applications, such real-time traces consist of not only runtime data but also user-interactions and changes to the GUI. Another requirement and maybe challenge for live programming for interactive GUI applications lies in visualizing several interconnected concepts during the execution: user-interactions, runtime data, and changes to the GUI. Most importantly, the traces must elucidate the *ordering* between all of these, as in existing live programming environments.

4.4 Existing First Steps and Remaining Challenges

There has been prior work on specific aspects of live programming for interactive GUI applications. First, there have been tools that deliver liveness for programming the initial UI state for interactive GUI applications within a programming environment [Burckhardt et al. 2013; Dey 2022a,b].

Then there is the Reactive Debugging Environment (RDE) [Schuster and Flanagan 2015] that enables live programming in JavaScript-based interactive GUI applications. RDE implements liveness by restricting JavaScript programs to have one rendering function and no function values in the global state. However, RDE does not explore visualizing real-time traces that present changes to the runtime data and the GUI during execution.

There are many JavaScript frameworks for developing web-based interactive GUI applications. Some come with hot reloading, which is a kind of liveness that maintains the state of the application as edits are made to the code without the need of reloading the application [Source 2022; Vue.js 2022]. However, like RDE [Schuster and Flanagan 2015], they do not provide real-time information regarding how the execution of code results in changes to the runtime data and the GUI.

The closest to fulfilling the requirements of live programming for interactive GUI applications (Sec. 4.3) is Lively4 [Lincke et al. 2017], a live programming environment for Web Components, which is a new abstraction mechanism for the Web. Different from all the tools described above, Lively4 live-visualizes UI state changes by recording and replaying user-interactions. Still, Lively4 only visualizes the final UI state caused by a recorded event sequence as opposed to the full timeline of UI state changes. The visual connection between each step of code execution to its corresponding UI state is missing in Lively4.

In summary, there are two remaining challenges yet to be addressed in all of the above work towards live programming for interactive GUI applications:

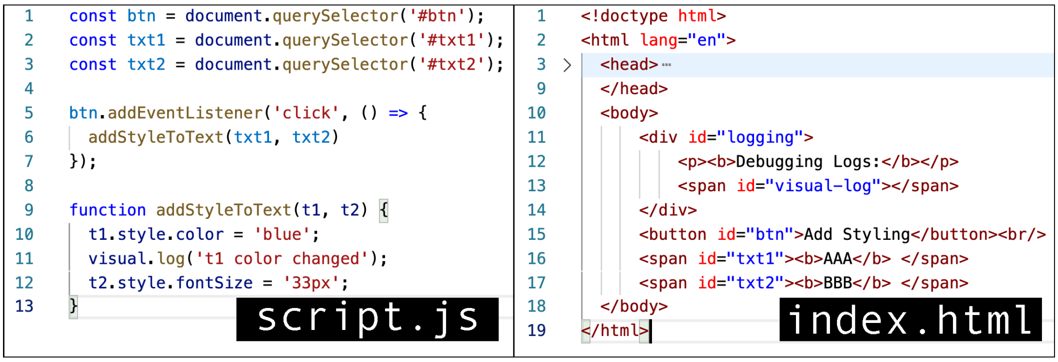
(1) Full-history liveness that visualizes the timeline of how the UI state changes over time; and
(2) The visual connection between each execution step and the UI changes for which it is responsible.

5 RECENT EFFORTS: LIVEUP

In this section, I discuss my recent work of a live programming experience for developing interactive GUI applications under the supervision of my advisor Sorin Lerner. Specifically, we propose a programming paradigm called *Live UI Programming* (LiveUP), and a programming interface called InteractLive that implements the LiveUP paradigm. Given a particular event sequence that only needs to be provided once, InteractLive gives the user the ability to fine-tune the behavior of an interactive GUI application with live feedback while also examining intermediate steps of execution. This section provides an overview of the design of InteractLive and a summary of the results from a qualitative evaluation of InteractLive.

5.1 Design

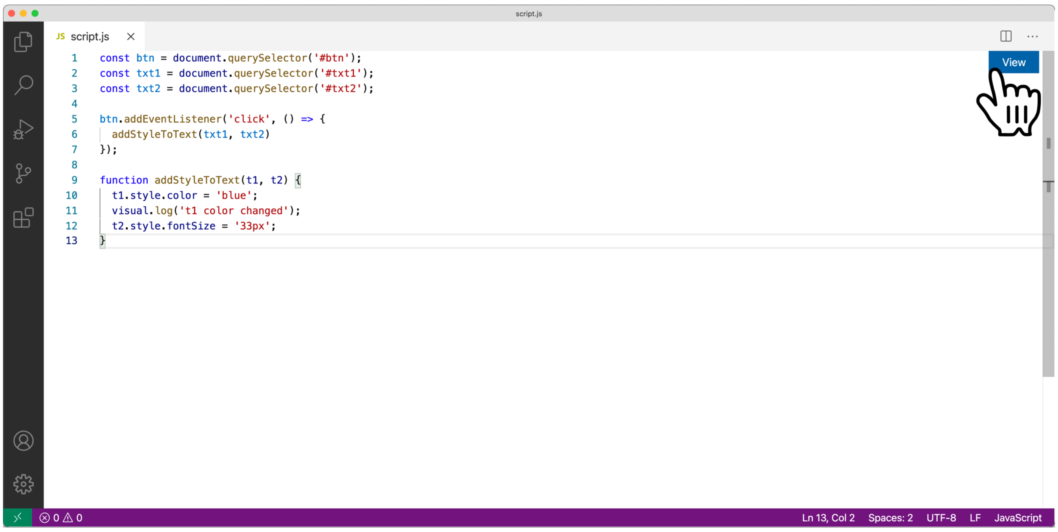
To demonstrate the design and interaction flow of InteractLive, we use a small application written in HTML/CSS and JavaScript, the source code of which is shown in Fig. 2. As a representative



```javascript
const btn = document.querySelector('#btn');
const txt1 = document.querySelector('#txt1');
const txt2 = document.querySelector('#txt2');

btn.addEventListener('click', () => {
  addStyleToText(txt1, txt2)
});

function addStyleToText(t1, t2) {
  t1.style.color = 'blue';
  visual.log('t1 color changed');
  t2.style.fontSize = '33px';
}
```

```html
<!doctype html>
<html lang="en">
  <head>···
  </head>
  <body>
      <div id="logging">
          <p><b>Debugging Logs:</b></p>
          <span id="visual-log"></span>
      </div>
      <button id="btn">Add Styling</button><br/>
      <span id="txt1"><b>AAA</b> </span>
      <span id="txt2"><b>BBB</b> </span>
  </body>
</html>
```

Fig. 2. The JavaScript (left) and HTML (right) code used for the demo in Fig. 3-Fig. 4.

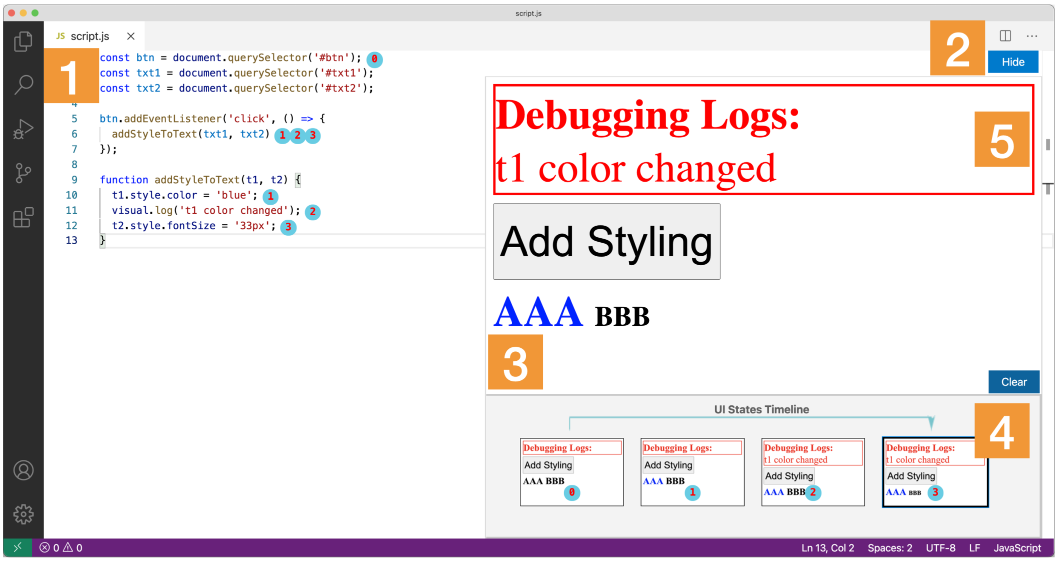


```javascript
const btn = document.querySelector('#btn');
const txt1 = document.querySelector('#txt1');
const txt2 = document.querySelector('#txt2');

btn.addEventListener('click', () => {
  addStyleToText(txt1, txt2)
});

function addStyleToText(t1, t2) {
  t1.style.color = 'blue';
  visual.log('t1 color changed');
  t2.style.fontSize = '33px';
}
```

Fig. 3. InteractLive, a live programming environment for developing interactive GUI applications written with JavaScript, when first started. To launch the InteractLive experience, the user can click on the "View" button (annotated with a cursor hand pointer icon).

example, consider a programmer Sam is trying to understand and modify the code for this application using InteractLive. Sam first sees Fig. 3, the initial InteractLive interface. As is shown, InteractLive bases on top of a full-featured IDE to provide live programming for developing interactive GUI applications. It supports local development of applications written in HTML/CSS and JavaScript without frameworks.

Sam starts using InteractLive by clicking on the "View" button (Fig. 3) to select the main HTML file of the application, `index.html` in Fig. 2. The file is then rendered in the view area at the right of the screen, and the button text is changed from "View" to "Hide". When necessary, Sam can toggle the view area on and off by pressing the "Hide"/"View" button. After skimming the source code, Sam roughly understands what the application does: a `click` event handler is attached to a DOM element with id `btn`, which should be the button with text "Add Styling", such that when the button is clicked, the styling of two text elements will change, and some text will be visual-logged. Sam is unsure of what the two text elements are, and what `visual.log` means. They decide to click on "Add Styling" in the interface, which gets them to Fig. 4. They see that the code in the editor is



```
const btn = document.querySelector('#btn'); 0
const txt1 = document.querySelector('#txt1');
const txt2 = document.querySelector('#txt2');

btn.addEventListener('click', () => {
  addStyleToText(txt1, txt2) 1 2 3
});

function addStyleToText(t1, t2) {
  t1.style.color = 'blue'; 1
  visual.log('t1 color changed'); 2
  t2.style.fontSize = '33px'; 3
}
```

script.js
1
2
Hide
Debugging Logs:
t1 color changed
5
Add Styling
AAA BBB
3
Clear
UI States Timeline
4
Debugging Logs: Add Styling AAA BBB 0
Debugging Logs: Add Styling AAA BBB 1
Debugging Logs: t1 color changed Add Styling AAA BBB 2
Debugging Logs: t1 color changed Add Styling AAA BBB 3
⊗ 0 ⚠ 0
Ln 13, Col 2 Spaces: 2 UTF-8 LF JavaScript

Fig. 4. InteractLive after the button with text "Add Styling" in the rendered application has been clicked: (1) a code editor annotated with UI-code connector labels; (2) the "Hide"/"View" button for toggling the view area on and off; (3) the application populated by `index.html` in Fig. 2 that the user could interact with as usual; (4) the UI States Timeline that shows the recorded and replayed `click` event on the "Add Styling" button; (5) the Visual Logging area built into the application UI that works like a simplified browser console.

annotated with some numbered circles (Fig. 4-(1)), the view area of the application (Fig. 4-(3)) now has a "UI States Timeline" on top (Fig. 4-(4)), and some text is added to the "Debugging Logs" area in the application (Fig. 4-(5)).

UI States Timeline. The "UI States Timeline" (Fig. 4-(4)) first attracts Sam's attention. Fig. 5 provides a closer look. The rectangles are the *UI states* produced by the mouse click that Sam just provided. After the `click` event, whenever there is a change to the entire UI page, a UI state is created. Sam notices that the leftmost UI state (Fig. 5-(a)) is the initial UI they saw before the button click. Sam further clicks on each UI state to examine its larger view in the view area. Now, they click on the rightmost UI state, and its larger view is presented in Fig. 4-(3).

Above the sequence of UI states, Sam sees a blue arrow spanning across them (Fig. 5-(b)), which indicates the `click` event they just provided to produce these UI states. The event arrow marks the start and the end UI states of the event, and in between are the intermediate UI states. Since Sam only provides one `click`, there is only one arrow. When there are multiple `click`s, there will be multiple arrows. Event arrows are blue except for arrows that start and end in the same UI state: we use the color of orange to highlight such arrows to indicate that the corresponding event for that arrow did not create new UI states (i.e., did not change the UI).

UI-code connector labels. Sam notices that there are numbered circles placed on the UI states (Fig. 5-(c)) as well as at the end of code lines inside the code editor (Fig. 4-(1)). They are *UI-code connector labels* that visually connect the code execution and the UI states. The semantics of such labels is defined as follows. Suppose we have a UI state annotated with a label i in the UI States Timeline, and there is a line of code annotated with the same label. This means two things: (1) right after the i-labeled line of code finished executing, the state of the UI was the one annotated with i in the timeline; (2) right before the i-labeled line of code started executing, the state of the UI was the one annotated with i in the timeline. With these labels (in Fig. 4), Sam quickly realizes that

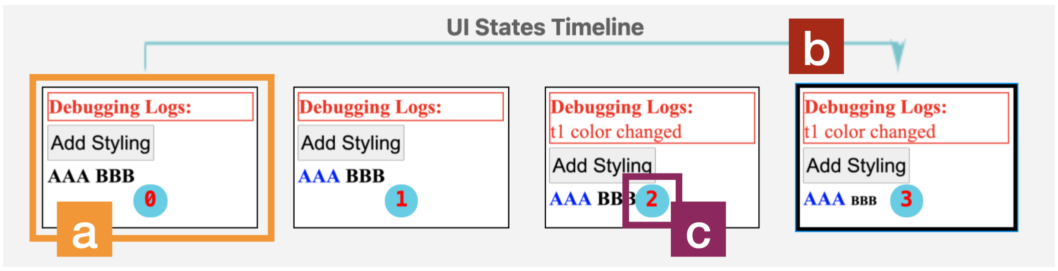


Fig. 5. The UI States Timeline of InteractLive that shows all the UI states caused by each recorded event: (a) one UI state that can be clicked on for examining its larger view in the view area (Fig. 4-(3)); (b) an event arrow that marks one `click` event as well as its starting and ending UI states; (c) a UI-code connector label that can also be found inside the code editor.

when they clicked on "Add Styling", in `script.js`, the `click` event handler at line 5 was executed, which further called `addStyleToText`. Calling `addStyleToText` created UI states 1, 2 and 3, which refers to changing the color of "AAA" (state 1) at line 10, visual-logging (defined below) a text to "Debugging Logs" (state 2) at line 11, and changing the font size of "BBB" (state 3) at line 12.

Visual logging. Calling `visual.log` logs its argument to "Debugging Logs", a visual logging area at the top of the application's interface (Fig. 4-(5)), and creates a new UI state for itself. Sam is familiar with logging to the browser console, and soon realizes that this "Debugging Logs" is just like a console built-into the application's UI, while `visual.log` is the `console.log` in this context. Indeed, `visual.log` can be regarded as a simplified version of `console.log` (it only takes one primitive as the argument), and the user can use it to examine non-UI state data values. For example, adding `visual.log(t1.style.color)` after line 10 will log "blue" to the visual logging area. Looking closer at UI states 1 and 2 in Fig. 5, Sam notices that the log at line 11 is not added to the interface until in UI state 2. Sam then realizes that users can use the UI States Timeline, focusing on the changes in the visual logging area, to reason about the sequence of data changes (results of visual-logging).

Live programming. After understanding the behavior of the application, Sam is ready to make some change to line 10: `t1.style.color = 'blue'` in `script.js`. From Fig. 4 they know that line 10 has a UI-code connector label 1. Sam then clicks on the UI state 1 in the timeline to see its larger view inside the view area. At this point, text "AAA" in that UI state is blue. Sam wants "AAA" to be green instead. They start typing to replace `blue` with `green`, and immediately sees the UI-code connector labels inside the code editor change from blue to gold. Meanwhile, Sam notices that the rest of the existing visualization stays unchanged, and realizes that as they are in the middle of typing, the code is no longer the same as before, and hence the InteractLive visualization is outdated and can be safely ignored. Once Sam finishes typing, they save the changes, and see the following updates in InteractLive within seconds without having to re-click on "Add Styling": (1) "AAA" in UI states 1 and onward and in the larger view (of the previously selected UI state 1) turns green; (2) the UI-code connector labels in the code editor have their color back to blue, with their positions unchanged.

What Sam experienced is live updates provided by InteractLive with the help of event recording (described below). Sam notices that the larger UI state presented in the view area is still UI state 1, what they were examining before the code change. Note that the number of UI states can differ depending on the code changes. As such, we need to consider which UI state gets its larger view to be presented in the view area, if the number of UI states resulted by the code changes is different from before. We decided that if the number of UI states remains the same after some code changes, then in the view area InteractLive still shows the (updated) larger view of the UI state that was previously selected. If the number of UI states changes after the update, InteractLive projects the larger view of the final UI state of the entire recorded event sequence to the view area, and automatically scrolls to that UI state in the UI States Timeline. Sam still sees the larger view of the previously selected intermediate UI state because the number of UI states remains unchanged before and after their code change.

In addition, live programming will only work when the entire program contains no syntax errors and the JavaScript execution leads to no runtime errors. In cases where the program has an error, InteractLive will report the error to an area overlaying the usual UI States Timeline area, and the rest of the visualization stays out-of-date. It is the programmer's responsibility to fix the error based on the reported error message before attempting another visualization update.

Event recording. As is described in previous paragraphs, the user can interact with the rendered application in the view area at the right part of the screen through mouse clicks as usual, and InteractLive will automatically record these mouse clicks. All the UI states produced by all the mouse clicks will be displayed in the UI States Timeline (Fig. 4-(4)), as will their corresponding UI-code connector labels and event arrows in the InteractLive interface. InteractLive records and replays one event sequence (which could include multiple events) at a time to repeatedly visualize the UI states created by the sequence, which further empowers live programming. Now, Sam is satisfied with what they have for the `click` event handling of "Add Styling". Moving onto adding more buttons and their event handlers, Sam then clicks on the "Clear" button (Fig. 4, on top of the UI States Timeline) to reset the UI of the application to its initial state and clear the recorded event sequence, ready to start from the beginning.

5.2 Users' Perceptions of LiveUP

LiveUP aims to live-visualize the connection between code execution and changes to the GUI and the runtime data in the context of user-specified interactions to the GUI application. As such, we would like to better understand users' perceptions of LiveUP for debugging interactive GUI applications. We conducted a 90-minute *qualitative* study with 12 participants experienced with developing JavaScript-based interactive GUI applications, in which each participant completed some debugging tasks using the InteractLive implementation of LiveUP and shared their perceptions of the tool with us. Through the study we found the following:

(1) Participants found LiveUP easy to understand and use;
(2) Participants considered LiveUP to be helpful for debugging interactive GUI applications as it provided visual aid for understanding code execution;
(3) Participants noted that LiveUP provided them with an innovative debugging experience for interactive GUI applications while also complementing existing paradigms.

A detailed description of the protocols of the study is outside of the scope of this report. The full manuscript on LiveUP, including the user study setup and results, is currently under review for publication but available upon request.

6 FUTURE DIRECTIONS

6.1 Encouraging Runtime Inspection with More Liveness

From our study, we observed two different ways in which participants used live programming for debugging interactive GUI applications through the InteractLive implementation of LiveUP.

In general, 9 out of 12 participants used InteractLive in the way that we expected. They started the tasks by first reading the instructions, then the source code, and finally interacting with the application rendered in InteractLive. They further used the visual information produced by

InteractLive to facilitate their understanding of the behavior of the event handlers invoked by their interactions. Once they figured out how and where the code went wrong, they started going back-and-forth between editing the source code and examining the live visual feedback in InteractLive until the end of the tasks.

However, three participants started the tasks differently. They first read the instructions, then the source code, but chose *not* to interact with the application immediately. They spent a noticeable period of time doing mental tracing (*understanding*) of the code rather than actively interacting with the application and using InteractLive (*inspecting*) to guide their tracing. They appear to only start providing user-interactions to the application *after* fully understanding the code (or at least getting a very good understanding of the code). At that point, they used InteractLive to quickly confirm/correct their understanding, and then quickly moved onto source code editing to fix the application's behavior while using InteractLive for its live feedback. We observed the same "reading lots, interacting little" behavior from these participants in the Baseline condition as well, where InteractLive was not used. Regardless of the condition, our task instructions recommended that after briefly reading the code, one can interact with the application to reinforce their understanding through the information provided by the given programming interface. Still, these participants decided to spend more time understanding the code through mental reasoning than inspecting.

Our observation that some participants relied on mental reasoning rather than active inspection to understand code is consistent with the work of Minelli et al. [Minelli et al. 2015], who revealed that during comprehension, programmers spent most of the time reading code (i.e., "staring at the screen") but less than 1% of that time using inspection activities to facilitate their understanding. As such, Minelli et al. recommended that programming environments should be improved to encourage more active program understanding, potentially through runtime inspection. This recommendation echoes with Kirsch and Maglio [Kirsh and Maglio 1994], who found that Tetris players performed actions all the time to both complete a goal (*pragmatic actions*) and understand the world they were in (*epistemic actions*). In addition, existing literature shows that program visualizations guide towards specific program comprehension strategies that make task completion faster and more accurate [Duru et al. 2013]. Therefore, one possible implication is that program visualizations encourage *epistemic actions* which help with code comprehension and should be more adopted in programming environments.

Our work is a step towards this direction by creating live visualizations in developing interactive GUI applications. However, our work still has limited liveness: liveness is not presented unless the user interacts with the application in the first place. One interesting direction would be to enable more liveness without requiring user-interactions, through mechanisms like automated event generation. With more liveness in the programming interface, users could be encouraged to spend more time actively inspecting the runtime behavior to understand the code more effectively.

6.2 When Live Programming Shows *How* to Fix the Code

In our study, two participants located the bugs of the given interactive application through InteractLive but did not know how to fix them, eventually failing the debugging tasks. We summarize two possible causes behind these debugging failures. First, not enough familiarity with API calls and/or CSS properties. Even if participants could search the web for documentation and resources for help, the web search requires an additional context switch between the programming environment and the browser and can take time. Second, programmers might have an expected UI state to reach in mind but do not know how to achieve it via code.

It is true that liveness can provide insights on why the application does not behave as expected, but it cannot repair the issue or suggest possible fixes. To facilitate code fixing, more help is needed beyond live programming. We believe program synthesis and live programming can improve the two scenarios above from two separate angles.

First, to use API calls/third-party libraries correctly, program synthesis within the editor is a promising solution that requires less context switch than a web search. The Copilot synthesizer [Git 2022] already performs such code suggestions inside your editor. However, an open problem remains how to confirm that the synthesized code is truly what you want. We hypothesize that when fixing event handlers for interactive GUI applications through code suggestions, LiveUP's immediate feedback on UI state changes *when new code is synthesized* can bring tremendous help: programmers can thus rely on liveness to pick and modify code suggestions. Second, to synthesize code given expected UI states, direct manipulation can be of great help and has been extensively explored [Hempel and Chugh 2022; Hempel et al. 2019; Schuster and Flanagan 2016]. Among existing work, using direct manipulation with live programming in the context of interactive GUI applications is still under-explored. The work by Schuster and Flanagan [Schuster and Flanagan 2016] does however show some early promising results towards this direction. We believe that when using direct manipulation to synthesize event handling code, the code-UI visual connections produced by LiveUP can show how the synthesized code is executed to create the intended UI state changes, so that the programmer can further fine-tune the code.

With either code suggestions or direct manipulation, live programming (specifically LiveUP) and program synthesis can improve the live programming experience in developing interactive GUI applications, in which liveness really shows not only where but *how* the code can be fixed.

7 CONCLUSION

In this report, through a review of existing tools for understanding interactive GUI applications and tools for debugging such applications, I identified the possible design space of tools for both understanding *and* debugging interactive GUI applications implied by existing tools. I then reviewed the history and development of live programming environments. I argued that live programming for interactive GUI applications would be a promising research direction because live programming (1) can reduce the *gulf of evaluation* in programming through its help in program comprehension and debugging, and (2) fits well into the potential design space of tools for understanding and debugging interactive GUI applications.

By reviewing existing efforts towards live programming for interactive GUI applications, I recognized open problems in this domain yet to be addressed. As such, I introduced our recent work in this domain that attempts to address these open problems, LiveUP, and preliminary results of how users perceived our work.

I call for two future research directions towards live programming for interactive GUI applications based on the LiveUP work and its user evaluation: enabling more liveness to encourage runtime inspection and introducing more program synthesis to live programming, which can further facilitate the comprehension and debugging of the dynamic behaviors of interactive GUI applications.

ACKNOWLEDGMENTS

I would like to thank my advisor Sorin Lerner for his advice and mentorship on my research, including this report and the LiveUP work. I would also like to thank the Programming Systems Group at UC San Diego for the feedback on the earlier iterations of LiveUP. Sec. 5 and Sec. 6, in full, are currently under review for publication of the material. The report author is the primary investigator and author of this material. The only co-author is their advisor, Sorin Lerner.

REFERENCES

2022. Cucumber - BDD Testing & Collaboration Tools for Teams. https://cucumber.io/

2022. Cypress - JavaScript End to End Testing Framework. https://www.cypress.io/

2022. GitHub Copilot · Your AI Pair Programmer. https://github.com/features/copilot

2022. Selenium. https://www.selenium.dev/

Abdulaziz Alaboudi and Thomas D. LaToza. 2021. Edit - Run Behavior in Programming and Debugging. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–10. https://doi.org/10.1109/VL/HCC51201.2021.9576170

Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2014. Understanding JavaScript Event-Based Interactions. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, Hyderabad India, 367–377. https://doi.org/10.1145/2568225.2568268

Apple Inc. 2022. Tools - Safari. https://developer.apple.com/safari/tools/

Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. 2016. Time-Travel Debugging for JavaScript/Node.Js. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Seattle WA USA, 1003–1007. https://doi.org/10.1145/2950290.2983933

Benjamin Biegel, Benedikt Lesch, and Stephan Diehl. 2015. Live Object Exploration: Observing and Manipulating Behavior and State of Java Objects. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 581–585. https://doi.org/10.1109/ICSM.2015.7332518

Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's Alive! Continuous Feedback in UI Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 95–104. https://doi.org/10.1145/2491956.2462170

Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*. ACM, St. Andrews Scotland, United Kingdom, 473–484. https://doi.org/10.1145/2501988.2502050

Brian Burg, Amy J. Ko, and Michael D. Ernst. 2015. Explaining Visual Changes in Web Interfaces. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, Charlotte NC USA, 259–268. https://doi.org/10.1145/2807442.2807473

Lautaro Cabrera, John H. Maloney, and David Weintrop. 2019. 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*. ACM, Boise, ID, USA, 227–236. https://doi.org/10.1145/3311927.3323138

Miguel Campusano, Alexandre Bergel, and Johan Fabry. 2016. 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*. ACM, Amsterdam, Netherlands, 8 pages. http://bergel.eu/MyPapers/Camp16-ComprehensionWithLRP.pdf

Pei-Yu (Peggy) Chi, Sen-Po Hu, and Yang Li. 2018. Doppio: Tracking UI Flows and Code Changes for App Development. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–13. https://doi.org/10.1145/3173574.3174029

Chrome Developers. 2022. Chrome DevTools - Overview. https://developer.chrome.com/docs/devtools/overview/

Frank Cornelis, Andy Georges, Mark Christiaens, Michiel Ronsse, Tom Ghesquiere, and Koen De Bosschere. 2003. A Taxonomy of Execution Replay Systems. In *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*.

Obert DeLine, Anyel Fisher, Adrish Chandramouli, Onathan Goldstein, Ichael Barnett, Ames Terwilliger, and Ohn Wernsing. 2015. Tempe: Live Scripting for Live Data. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Atlanta, GA, 137–141. https://doi.org/10.1109/VLHCC.2015.7357208

Robert DeLine and Danyel Fisher. 2015. Supporting Exploratory Data Analysis with Live Programming. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Atlanta, GA, 111–119. https://doi.org/10.1109/VLHCC.2015.7357205

Robert A DeLine. 2021. 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*. ACM, Yokohama Japan, 1–11. https://doi.org/10.1145/3411764.3445267

Ritwick Dey. 2022a. VSCode Live Server. https://github.com/ritwickdey/vscode-live-server

Ritwick Dey. 2022b. VSCode Live Server++. https://github.com/ritwickdey/vscode-live-server-plus-plus

Hacı Ali Duru, Murat Perit Çakır, and Veysi İşler. 2013. How Does Software Visualization Contribute to Software Comprehension? A Grounded Theory Approach. *International Journal of Human–Computer Interaction* 29, 11 (Nov. 2013), 743–763. https://doi.org/10.1080/10447318.2013.773876

John D. Gould. 1975. Some Psychological Evidence on How People Debug Computer Programs. *International Journal of Man-Machine Studies* 7, 2 (March 1975), 151–182. https://doi.org/10.1016/S0020-7373(75)80005-8

Paul A. Gross, Micah S. Herstand, Jordana W. Hodges, and Caitlin L. Kelleher. 2010. A Code Reuse Interface for Non-Programmer Middle School Students. In *Proceedings of the 15th International Conference on Intelligent User Interfaces - IUI '10*. ACM Press, Hong Kong, China, 219. https://doi.org/10.1145/1719970.1720001

Christopher Michael Hancock. 2003. *Real-time programming and the big ideas of computational literacy.* Thesis. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/61549

Brian Hempel and Ravi Chugh. 2022. *Maniposynth: Bimodal Tangible Functional Programming.* Technical Report. https://doi.org/10.4230/LIPIcs.ECOOP.2022.16 arXiv:2206.14992 [cs]

Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology.* 281–292. https://doi.org/10.1145/3332165.3347925 arXiv:1907.10699 [cs]

Joshua Hibschman and Haoqi Zhang. 2015. 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 (UIST '15)*. Association for Computing Machinery, New York, NY, USA, 270–279. https://doi.org/10.1145/2807442.2807468

Joshua Hibschman and Haoqi Zhang. 2016. Telescope: Fine-Tuned Discovery of Interactive Web UI Feature Implementation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. Association for Computing Machinery, New York, NY, USA, 233–245. https://doi.org/10.1145/2984511.2984570

Ruanqianqian (Lisa) Huang, Kasra Ferdowsi, Ana Selvaraj, Adalbert Gerald Soosai Raj, and Sorin Lerner. 2022. 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 (SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 495–501. https://doi.org/10.1145/3478431.3499305

Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 1985. Direct Manipulation Interfaces. *Human–Computer Interaction* 1, 4 (Dec. 1985), 311–338. https://doi.org/10.1207/s15327051hci0104_2

Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. Association for Computing Machinery, New York, NY, USA, 737–745. https://doi.org/10.1145/3126594.3126632

Saketh Ram Kasibatla. 2018. *Seymour: A Live Programming Environment for the Classroom.* Ph. D. Dissertation. UCLA.

David Kirsh and Paul Maglio. 1994. On Distinguishing Epistemic from Pragmatic Action. *Cognitive Science* 18, 4 (1994), 513–549. https://doi.org/10.1207/s15516709cog1804_1

Amy J. Ko and Brad A. Myers. 2009. Finding Causes of Program Output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, Boston MA USA, 1569–1578. https://doi.org/10.1145/1518701.1518942

J. Kramer, J. Kurz, T. Karrer, and J. Borchers. 2014. How live coding affects developers' coding behavior. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* 5–8. https://doi.org/10.1109/VLHCC.2014.6883013 ISSN: 1943-6106.

Sorin Lerner. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3313831.3376494

Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions about Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. Association for Computing Machinery, New York, NY, USA, 2481–2490. https://doi.org/10.1145/2556288.2557409

Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a Live Development Experience for Web-Components. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience.* ACM, Vancouver BC Canada, 28–35. https://doi.org/10.1145/3167109

Renée McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: A Review of the Literature from an Educational Perspective. *Computer Science Education* 18, 2 (June 2008), 67–92. https://doi.org/10.1080/08993400802114581

Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *2015 IEEE 23rd International Conference on Program Comprehension.* 25–35. https://doi.org/10.1109/ICPC.2015.12

Mozilla. 2022. Firefox DevTools User Docs — Firefox Source Docs Documentation. https://firefox-source-docs.mozilla.org/devtools-user/

Brad Myers, Alan Ferrency, Rich McDaniel, and Roger Dannenberg. 1996. Debugging Interactive Applications. (Jan. 1996). https://doi.org/10.1184/R1/6621842.v1

Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 2020. Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3426428.3426919

Donald A. Norman. 1988. *The Psychology of Everyday Things*. Basic Books, New York, NY, US. xi, 257 pages.

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–32. https://doi.org/10.1145/3290327

Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 105–108. https://doi.org/10.1109/VLHCC.2009.5295287

David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-Style Programming. *The Art, Science, and Engineering of Programming* 3, 3 (Feb. 2019), 9:1–9:39. https://doi.org/10.22152/programming-journal.org/2019/3/9

Christopher Schuster and Cormac Flanagan. 2015. Live Programming for Event-Based Languages. In *Proceedings of the 2015 Reactive and Event-based Languages and Systems Workshop, REBLS*, Vol. 15. https://users.soe.ucsc.edu/~cormac/papers/15rebls.pdf

Christopher Schuster and Cormac Flanagan. 2016. Live programming by example: using direct manipulation for live program synthesis. In *LIVE Workshop*. https://chris-schuster.net/live16/live16-lpbe.pdf

Emmanuel Senft, Michael Hagenow, Robert Radwin, Michael Zinn, Michael Gleicher, and Bilge Mutlu. 2021. Situated Live Programming for Human-Robot Collaboration. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. ACM, Virtual Event USA, 613–625. https://doi.org/10.1145/3472749.3474773

Facebook Open Source. 2022. react-refresh. https://www.npmjs.com/package/react-refresh

Steven L. Tanimoto. 1990. VIVA: A Visual Language for Image Processing. *Journal of Visual Languages & Computing* 1, 2 (June 1990), 127–139. https://doi.org/10.1016/S1045-926X(05)80012-6

Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *2013 1st International Workshop on Live Programming (LIVE)*. 31–34. https://doi.org/10.1109/LIVE.2013.6617346

Bret Victor. 2012. Learnable Programming. http://worrydream.com/LearnableProgramming/

A. Von Mayrhauser and A.M. Vans. 1995. Program Comprehension during Software Maintenance and Evolution. *Computer* 28, 8 (Aug. 1995), 44–55. https://doi.org/10.1109/2.402076

Vue.js. 2022. Hot Reload | Vue Loader. https://vue-loader.vuejs.org/guide/hot-reload.html#state-preservation-rules

Wengran Wang, Gordon Fraser, Mahesh Bobbadi, Benyamin T. Tabarsi, Tiffany Barnes, Chris Martens, Shuyin Jiao, and Thomas Price. 2022. 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, 1–10. https://doi.org/10.1109/VL/HCC53370.2022.9833105

E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. 1997. Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems?. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*. ACM, Atlanta Georgia USA, 258–265. https://doi.org/10.1145/258549.258721

Xiong Zhang and Philip J. Guo. 2017. 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*. ACM, Québec City QC Canada, 691–702. https://doi.org/10.1145/3126594.3126663

Chunqi Zhao, I-Chao Shen, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2022. ODEN: Live Programming for Neural Network Architecture Editing. In *27th International Conference on Intelligent User Interfaces (IUI '22)*. Association for Computing Machinery, New York, NY, USA, 392–404. https://doi.org/10.1145/3490099.3511120