2025 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) | 979-8-3315-0233-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/VL-HCC65237.2025.00022

An Information Foraging Interpretation of Liveness

1st Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
0000-0001-9454-8381
patrick.rein@hpi.de

2nd Stefan Ramson
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
0000-0002-0913-1264
stefan.ramson@hpi.de

3rd Tom Beckmann
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
0000-0003-0015-1717
tom.beckmann@hpi.de

4th Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
0000-0002-4249-6003
robert.hirschfeld@hpi.uni-potsdam.de

***Abstract*—Various programming environments feature live programming as a central part of their programming experience. Liveness in these environments is analyzed by several theories, which describe the different forms of liveness and how they influence the programming experience. Unfortunately, these are isolated theories only concerned with live programming features. They can not explain liveness in terms of general interactions with programming tools.**

One theory that has successfully explained how programmers interact with various kinds of tools is information foraging theory (IFT). In this paper, we propose an IFT interpretation of liveness. By interpreting liveness in terms of a more comprehensive theory of programmer behavior, we can explain the role of liveness in common software development activities such as debugging.

We have explored our IFT interpretation of liveness in a first controlled experiment. We investigated the hypothesis that when we remove liveness from live dynamic introspection tools, programmers will use them less often, as using them has a higher cost. We have conducted a post hoc analysis of the results for which we used the edit-run-cycle model adapted to analyze the usage of live programming tools. The post hoc results hint that programmers did shift away from introspection tools. While our study only features post hoc results, it suggests that the IFT interpretation of liveness is a fruitful avenue.

***Index Terms*—live programming, liveness, information foraging theory, programming tools, controlled experiment, edit-run cycle**

I. Introduction

Liveness in programming tools is the impression of changing a program while it is running [1]. Various tools support liveness, including commercial programming systems, such as MS Excel and Jupyter Notebooks. Tool designers assume that liveness improves the programming experience, and early experimental evidence hints that it can indeed improve programmer productivity under certain circumstances [2], [3].

Typically, liveness in programming tools is motivated by design frameworks or principles such as Tanimoto's liveness levels that describe how many discrete steps are involved in accessing dynamic information [4], [5], the steady frame [6], with a stable "frame" of relevant variables whose values are "steadily" available to the programmer, or immediacy through the reduction of the temporal, spatial, and semantic distance between cause and effect [7], [8]. While these principles can guide tool designs through visions of how liveness may affect the programming experience, they do not suffice to explain the mechanism by which liveness helps programmers. In particular, these principles do not suffice to predict how liveness affects the way programmers work or how liveness interacts with other mechanisms.

In this paper, we present an interpretation of live programming features through the lens of the information foraging theory (IFT) [9], which has been repeatedly used to successfully model effects of programming tools [10]–[12]. Originally, IFT was used to model users' web search behavior. IFT is based on the idea that participants navigate a topology of information artifacts by choosing the next information artifact based on a cost-benefit estimate. IFT has been successfully used to predict and describe programmers' behavior for various programming tools and activities, including static navigation tools and debugging [10]–[12]. By describing the effects of liveness through information foraging theory, we embed live programming in a larger theory of programmer behavior. In the course of this, we also propose a detailed IFT interpretation of programming tools working with dynamic information, as live programming is mostly concerned with dynamic information, and the existing literature does not provide a sufficiently detailed interpretation yet. With our interpretation of live programming in terms of IFT, researchers can make predictions of how live programming features would impact programmer behavior in novel tools or contexts.

To test our IFT-interpretation of liveness, we designed a first controlled experiment focusing on the duration of feedback loops. With this experiment, we tested the hypothesis that an increase in the access time of introspection tools would cause programmers to shift away from them. We applied a post hoc analysis based on the edit-run cycle model [13], [14], which is used to analyze how programmers alternate between working with the code (edit) and observing program behavior, either through surface behavior or run-time state (run).

We needed to adapt the edit-run cycle model, as it was previously mostly used to analyze conventional programming environments. Thus, the code book and metrics did not match our context and research question. Therefore, we adapted the model to accommodate interactions that are possible within live programming environments and derived a metric to measure interactions with live introspection tools. The results of

This work was supported by Deutsche Forschungsgemeinschaft (DFG) grant #449591262, SAP, and the HPI–MIT "Designing for Sustainability" research program.

1943-6106/25/$31.00 ©2025 IEEE
DOI 10.1109/VL-HCC65237.2025.00022

this post hoc analysis suggest that programmers do indeed shift away from dynamic introspection tools in the presence of delays in the sense that they spend less net time with the tools. The analysis also showed that the main analysis failed because the interaction events that we had collected for the analysis contained too much noise, necessitating manual coding as described in Section III-E.

In summary, the contributions of this paper are:

- a detailed information foraging theory interpretation of dynamic introspection tools, and
- post hoc results from a first experiment on the information foraging interpretation of the effects of liveness.

A. Outline

In the following, we first outline our understanding of IFT and our proposed detailed IFT interpretation of dynamic introspection tools and liveness's role in it (Section II). We then outline the experiment design of a first experiment investigating our IFT interpretation of liveness (Section III and Section IV). Finally, we discuss the experiment results and their implications (Section V).

II. Integration of Liveness into IFT Model

IFT has successfully been used to predict how programmers use programming tools [10]–[12], [15]. Nevertheless, previous IFT interpretations of programming tools have focused on static tools and their features. We propose a detailed IFT interpretation of dynamic introspection tools and how liveness may affect their usage. This detailed IFT interpretation can serve as a foundation to revisit existing experiments on the impact of liveness.

A. Information Foraging Theory

Information Foraging Theory (IFT) [9], [10] aims to describe how people seek information in a graph of information artifacts. It successfully modelled users' behavior for hypermedia content on the web [16], but has since also been successfully applied to other domains, including programming tools [10]–[12], [15].

IFT has two core concepts: the *information topology* and the *user*.

The information topology is a graph that consists of *information patches* (for instance, a source file or a documentation page) and *links* to navigate between these patches (for instance, a way to navigate to the definition of a method or a hyperlink to an extended documentation page). The information contained in a patch is represented as a set of *information features*. Information features that relate to the outgoing links of the patch are called *cues* (for instance, the text of a hyperlink or the method selector in a method call).

Users now try to fulfill an *information goal*, which is a set of desired information features. Users navigate through the information topology, whereby they can only look at one patch at a time, and navigating via links is associated with some cost. Based on these constraints, users try to fulfill their information goal by either a) exploiting a patch by consuming information features (for instance, reading the lines of a method), b) switching to a different patch, or c) *enriching* the topology with new patches or links.

Based on these two concepts, IFT now aims to describe the users' navigation behavior through the topology. When users decide to navigate to a different patch, they have to choose one of the outgoing links. IFT proposes that users choose the link with the strongest *scent*, which is the perceived probability that the patch at the other end of the link either contains desired information features or brings them closer to a patch that does.

Based on this informal description of users' behavior, the original authors also proposed a more formal model to predict user behavior [9]. The following is a very abbreviated form of the model detailing the core mechanism [10]. Within their information search, users want to make decisions that maximize the ratio of information value gained with regard to their information goal (V) per interaction cost (C): $max(V/C)$.

The costs are associated with consuming information features within a patch, following links, and enriching the topology. These costs (for instance, time to follow a link, time to consume an information feature) and properties of the topology (for instance, number of information features per patch, percentage of valuable patches overall) determine the upper bound for users' efficiency. However, users do not have perfect information on the actual value gained from their interactions and the actual costs, so they can only optimize the ratio of their estimates: $max(E(V)/E(C))$.

The efficiency of users now also depends on how well they can estimate the value and costs of interactions [9].

B. IFT Interpretation of Dynamic Introspection Tools

Dynamic introspection tools are helpful for programmers whenever they need to understand program behavior in detail, for instance during debugging, when behavior does not meet their expectations, or reverse engineering, when they have no clear idea yet of the general behavior of the program. The information goal in these circumstances is to understand behavior, thus to collect code locations, selected run-time states, and to derive the control flow. In the context of debugging, these elements of the information goal match the basic elements of explaining a failure: defect in the code, state infection, and infection chain [17].

Existing IFT interpretations of programming tools do not cover dynamic information and often focus on static programming tools [10]. While some studies have applied IFT to activities that involve dynamic information, most notably debugging [11], [12], most of these studies also focus on static information and regard dynamic information only as a single kind of patch without a more detailed interpretation or only as a way to enrich static information [18]. To enable a more coherent modelling of introspection tool usage, we propose a detailed IFT interpretation of them. Our interpretation covers dynamic introspection tools that allow programmers to explore run-time state in-time, such as object inspectors and stepwise debuggers. Over-time perspectives, such as call trace visualizations, are not covered by our interpretation yet.

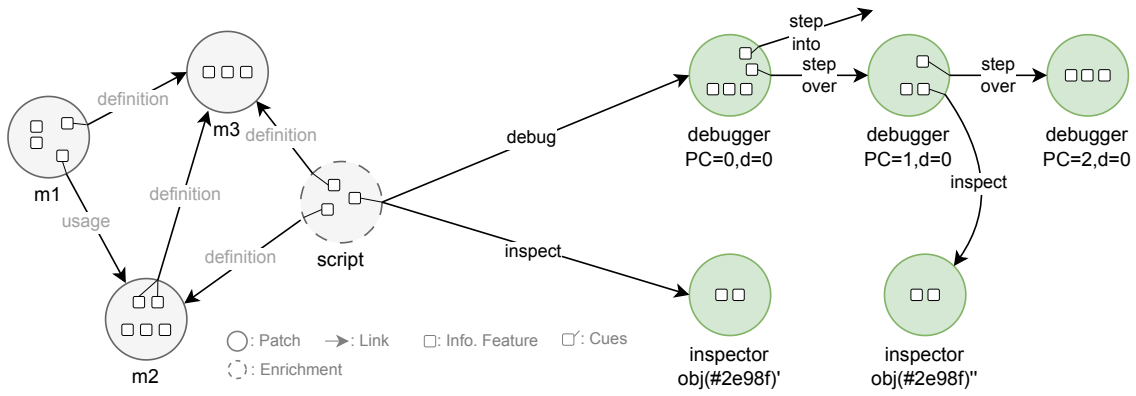

Fig. 1. A visualization of an information foraging topology of programming artifacts. Gray patches represent static patches, such as methods (m1 - m3) and a script. Green patches represent dynamic patches, such as different states in the debugger, and an object at two different points in time.

We capture dynamic information as patches that are defined by a certain view (for instance, object inspector or stepwise debugger) and a moment in time in the execution of a program (see Figure 1). Information features within this patch are then all information available in that view (for example, instance variables in the object explorer; stack, PC, variables in the debugger). The links from such a dynamic information patch are the ones that open or change the view (e.g., expanding the object subtree in an instance variable; switching stack frame) or move the view to another moment in time (e.g. stepping or continuing in the debugger). As with other kinds of information, the summary information available in the view serves as cues for these links (e.g., print string of objects in instance variables; method selectors in code or stack frames) Finally, we regard writing a script to run the program as enrichment that creates a new script patch [12]. This script patch then has run links that allow programmers to navigate to dynamic information patches (e.g., by running it and opening the result in the object inspector; by running the script in the debugger).

Given this IFT interpretation of dynamic information, dynamic introspection tools can help programmers in three ways. Dynamic tools can improve the value of information, as it is more likely part of the infection chain than information deduced from static information (which might not actually be executed in the behavior in question). For instance, in a debugger, each method is guaranteed to be in the trace. Further, introspection tools can reduce the cost of consuming information features, as relevant information can be observed instead of needing to be deduced (for instance, example values for variables can be observed instead of being deduced from static data flow). Finally, when deciding how to proceed during an investigation of a behavior, introspection tools align users' estimate of a value closer to the real value, as observable runtime state can help decide which method or object might also be relevant.

This fine-grained IFT interpretation of dynamic introspection tools also opens up interesting opportunities to analyze introspection tool usage in detail. For instance, this lens gives a coherent description of one of the main inconveniences in a typical stepwise debugger. When debugging, step-over is a cheap navigation to the next patch, which is the state at the next PC shown in the debugger. However, this still requires sequential navigation interactions along the step-over links to get to the desired patch (state in the debugged process). For each navigation interaction, programmers have to evaluate relevant cues to decide whether they want to step into or step over an instruction. The cues comprise the current method's signature, the PC, and potentially run-time state and debug watches. This is not necessary for static navigation, as we can often directly jump to the desired location (for example via hyperlinks in the source code) or select it from a list (for instance in a method selector list for a class). Thus, we would expect debuggers to be improved by enabling navigation based on more useful links beyond "next run-time state" and based on more helpful cues (for instance, a list of method signatures). Indeed, projects such as the Whyline demonstrate that this interpretation holds [19]: when programmers gained access to "why" and "why not" questions regarding program execution, which navigated to relevant places in the source code, they were able to more quickly narrow down causes of bugs.

C. An IFT Interpretation of Liveness

Programmers benefit from dynamic introspection tools when working to understand a specific system behavior, as they offer patches with a higher value for the information goal (see Section II-B). In addition to this increased value, liveness now reduces the cost of using introspection tools by shortening the temporal, spatial, and semantic (interaction) distance [8] (see Figure 2).

The main way in which liveness reduces the cost of using introspection tools is by reducing the cost of navigating between static and dynamic patches.

Mainly, liveness reduces the cost of navigating from static to dynamic information. For instance, in the Squeak/Smalltalk

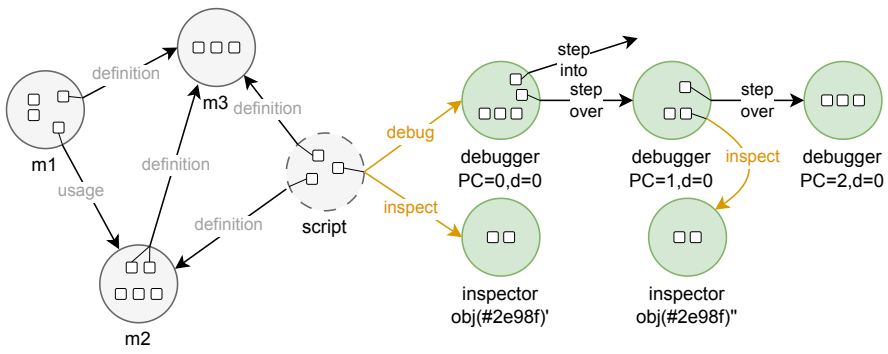

Fig. 2. A visualization of the impact of liveness on the links in a information topology. One effect of liveness is that it reduces the cost of accessing introspection tools, visualized by shorter distances between nodes.

live programming environment, programmers can select and execute any code via keyboard shortcuts and thus quickly get from a piece of code to either a debugger or an object inspector showing the result of the execution. In example-based live programming environments, functions are repeatedly executed on user-provided example input, and fine-grained run-time data is displayed directly next to code elements, such as statements or expressions [20]–[22]. Thus, they merge static and dynamic information into a single view, which reduces the cost of accessing dynamic information from the cost of navigating between different patches to the cost of consuming different kinds of information features in the same patch.

Several mechanisms also decrease the cost of navigating from dynamic to static information. For instance, in the YinYang environment, programmers can select a terminal output and jump to the source code that generated it [20], and in the Self environment, programmers can select user interface elements of a running application and navigate to the code that created it [23].

Finally, we conjecture that when liveness reduces the cost of accessing dynamic information consistently, it also reduces users' estimates of the cost consistently, thereby making the usage of dynamic information more likely in general.

D. Existing Studies of the Effects of Liveness through the Lens of IFT

To date, there are only a few controlled experiments on the effects of liveness. Of the four controlled experiments we are aware of, two focus on understanding programs [3], [24] and two on writing programs [25], [26]. While IFT is not directly concerned with creating programs, our IFT interpretation still allows us to relate their results through the same framework.

One experiment investigated the impact of feedback modes (*automatic* and *self-selected, delayed*) on debugging performance in a spreadsheet environment [2], [24]. They had 29 participants with considerable software development experience but no experience with the environment. Debugging performance was measured as the number of defects repaired in 15 minutes. Participants received two tasks: one about debugging the control logic of a seven-segment display and another about debugging a validation formula for hashes. The experiment did not find a significant overall difference in debugging performance, but found a significant difference for the hash validation task in isolation.

Interpreting the results with IFT, we argue that the tasks were too simple. When tasks are very simple, the information goal shifts from the infection chain to only the code locations, as they can be sufficient to understand the failure statically [27], [28]. As a result, the *value of using introspection tools* is lower, in particular in a spreadsheet environment, as they do not support the programmers much in their information goal. Still, the feedback mode significantly influenced programmer behavior, with the automatic feedback group making more changes and making changes earlier. As changes trigger updates of dynamic information in the environment, this higher number might result from the automatic feedback reducing the *cost for accessing dynamic information*.

Another experiment investigated the impact of the availability of live introspection tools on the debugging performance for simple and complex tasks [3]. The experiment had 37 third-year undergraduate students as participants with considerable experience in the programming environment used. In both conditions, participants had access to a debugger, but only in the live condition, they had access to a detailed object inspector, graphical meta-menus, and ubiquitous dynamic code evaluation. Debugging performance was measured as the time to successfully repair a defect. Participants worked on up to eight tasks. Simple tasks were bugs of commission and complex tasks were bugs of omission, so simple defects could be spotted, while for complex tasks, the behavior needed to be understood in detail to identify the missing part. The experiment found an overall significant difference in performance.

From an IFT perspective, the complex tasks ensured that the information goal was the infection chain. Further, as participants were experienced with the programming environment, they did not experience an additional cost of accessing the introspection tools due to not knowing how to use them.

One of the experiments on how liveness affects program

creation tested whether feedback modes (automatic, self-selected, no feedback) influence the programming effectiveness of novice programmers [25], [29]. The experiment was conducted in a newly developed live programming environment combined with a live algorithm visualization tool and a newly created programming language. The participants were 57 students from an introductory programming course. Programming effectiveness was measured as the number of correctly solved tasks. The three tasks asked participants to create basic algorithms, such as replacing and counting the occurrences of the number zero in an array. The experiment did not find a significant difference in programming effectiveness between feedback modes.

From an IFT perspective, we argue that the absence of an observable difference resulted from a high cost of using dynamic introspection tools due to the participants' low programming experience. Programming novices often have less (relative) experience in using introspection tools than experienced programmers [30]. Thus, using them requires more effort and thus results in higher cost or at least a higher cost estimate. Additionally, this might also lower the value gained from the tools, as the novices might not be able to get the information they want from them.

Finally, another experiment on how liveness affects program creation tested whether the availability of live feedback (*live* and *no feedback*) affected program creation effectiveness [26]. The experiment was conducted in a newly developed programming environment for JavaScript. There were ten participants with considerable programming experience but no experience with the new environment. Programming effectiveness was measured as the time to complete a program task. The three tasks were moderately complex. For instance, one task required participants to parse an RSS feed using a given library. The experiment did not find a significant overall difference in the task completion times between the live and no-feedback groups. However, for one task, there was a significant improvement for the group with live feedback, which had a strong effect. Also, the group with live feedback adopted a workflow of interleaved code creation and inspection of run-time state, while the group without live feedback used a sequential workflow.

From an IFT perspective, the changes in workflow match our interpretation of liveness as a cost reduction for navigation between static and dynamic information. Thus, programmers more often choose dynamic information when they deem it valuable, as cost is not the primary driver of their decision anymore. As this only led to improved effectiveness for some tasks, the overall effect on programming effectiveness seems to depend on how relevant the information goals are to the overall task.

III. Experimental Design

To investigate our IFT interpretation of liveness, we devised a first experiment designed to test a basic implication of it[1].

[1]The material for reproducing the experiment is available at https://doi.org/10.5281/zenodo.16740102.

A. Hypothesis

The hypothesis for this experiment was: "a higher access time reduces the usage of dynamic introspection tools for programmers who are experienced with live programming tools when they debug complex tasks."

A core element of our IFT interpretation of liveness is the fact that liveness decreases the cost of navigating from static to dynamic tools. Therefore, the presence of liveness should increase the usage of dynamic introspection tools, given that programmers appear to regard dynamic information as helpful. While there are no direct studies on whether programmers regard dynamic information as helpful, the findings of several studies in combination suggest so. Evaluation studies of dynamic tools find that programmers report that they find them helpful [31]. Beyond reporting that they found it helpful, programmers also used it in several studies on general programmer workflows [32], [33]. Finally, studies on novice and experienced programmers find that more experienced programmers often access dynamic information more often, suggesting that it is a rewarding strategy [34], [35].

We want to test how having and removing liveness from dynamic introspection tools changes how often programmers use them. But, as the experience of liveness can be created via spatial, temporal, or semantic proximity [8], we needed to decide on one type of proximity to test. We decided to test the influence of access duration (live, delayed) within live introspection tools. A previous experiment [3] has shown that the live introspection tools (for instance, live object inspector or ubiquitous code evaluation for inspection) in an exploratory-style live programming environment improve debugging efficiency. As programmers benefited from these tools, they have a high value, which is a prerequisite for observing changed usage due to cost (if the value was low to begin with, even lower costs might not motivate programmers to use them). Further, while they have been shown to benefit programmers, it is yet unclear whether this is because of their feature set or because they are live.

To ensure that there is the possibility that users change their usage of introspection tools, the experiment's scenario should make introspection tools useful. Thus, we sought debugging tasks that are complex enough that the participants have the goal of understanding (parts of) the infection chain (see Section IV-A).

Finally, we focused on programmers with prior experience with live programming tools to prevent spurious influences on the cost from lack of experience as seen in previous experiments (see Section IV-B).

B. Experiment Layout

We conducted the experiment as a within-subject design with one independent variable *access duration* (IV) and one dependent variable *tool usage frequency* (DV) (Sections III-D and III-E).

We also identified two major confounding variables: task complexity and experience of participants with liveness. We aimed to control both (Sections IV-A and IV-B).

C. Programming Environment

We briefly outline the programming environment used to introduce the tools and mechanisms affected by the duration of access.

We conducted the experiment in the object-oriented Squeak/Smalltalk exploratory live programming environment [36]. Programmers working in Squeak/Smalltalk do not modify a program but instead the object-graph of a running system. They determine the behavior of the system by modifying meta-objects such as class and method objects. We decided on the Squeak/Smalltalk environment as it features live introspection tools that are mature and, at the same time, variants of common tools found in other environments (live object inspector, stepwise debugger, graphical meta-menu). We argue that the environment is well-suited for the experiment, as liveness is at the core of most programming tools but as the tools are similar to common tools, we expect the insights to be transferable to other environments.

In detail, Squeak/Smalltalk features the following live introspection tools that are relevant for this experiment:

- a stepwise debugger that supports edit-and-continue with instant compilation and restarting of methods; also, any running process in the environment can be stopped at any moment and inspected in the debugger,
- ubiquitous code evaluation of selectable text: in the debugger against the state of the debugged process, in object inspectors against inspected object, in special workspace tool against local state of tool, in the code editor against class instance,
- graphical object inspectors that can be opened on any object and refresh periodically, and
- the "Halo" graphical meta-menu [37] to inspect the object-structures representing visible user interface elements and navigate to the underlying class object.

D. Operationalization of Access Duration (IV)

We distinguished between two access durations: *live* and *delayed*. In the *live* condition, programmers could use the live introspection tools directly without any modifications. In the *delayed* condition, programmers experienced a delay in some interactions with live introspection tools. The delay varied between eight and twelve seconds and was added to all interactions with live introspection tools that would provide the participants with new information. This involved the opening of any live introspection tool (object inspectors, Halo, stepwise debugger), interactive code evaluation, as well as navigating in the tree view of the object explorer, as opening a sub-tree is equivalent to opening another object inspector. Selecting user interface elements and other navigation interactions was not delayed.

During the delay, we blocked all interactions and showed a small widget in the middle of the screen saying "Copying image state…". To prevent social desirability effects resulting from participants guessing that the delay was the independent variable, we told participants beforehand that there was some analysis running in the background and that the dynamic behavior of some tasks interacted with that analysis, leading to longer delays when using introspection tools. To keep up this framing, we also showed the small widget in the *live* mode but only very briefly ($< 150\,\mathrm{ms}$).

E. Edit-Run Cycle Model Operationalization of Tool Usage (DV)

For our post hoc analysis[2], we operationalized tool usage based on the edit-run cycle model for programmers' interactions with programming environments, as it allows for further analysis beyond differences in tool usage [13], [14]. In detail, we operationalized tool usage as the ratio of programmers' time spent using introspection tools in relation to the overall time they spent investigating program behavior dynamically.

The edit-run cycle model was originally used to analyze how programmers alternate between working with the code (edit) and observing program behavior, either through surface behavior or run-time state (run). The model distinguishes between six activities of which five are relevant for our analysis: *browsing* a file, *editing* a file, *testing* program (observe surface behavior), *inspecting* program (observe run-time state), and *miscellaneous* activities (e.g., window management, editing search strings). The model then distinguishes between *edit* and *run* phases [13, Fig. 3]. An *edit* phase is a series of browse and edit activities; a run phase is a series of test and inspect activities. A series of activities is not interrupted by miscellaneous activities. The duration of a phase is the sum of the durations of the corresponding activities (excluding misc. activities).

As we were primarily interested in the usage frequency of introspection tools, we slightly adapted the original code book. In accordance with the original model, we classified using an introspection tool as an *inspect* activity. The main difference is that we classified observing logging output of the program as a *testing* activity instead of as an *inspect* activity. Further, the original code book assumed a programming environment in which programmers explicitly start a program for testing or debugging. As live environments do not distinguish between these modes, we further needed to adapt the definitions of a test activity. We define a *test* activity as an activity in which no *inspect* tools were used. As a result of these adaptations, we can deduce the time spent using introspection tools directly from the time spent with inspect activities. For all activities, we deducted the duration of any delays that occurred during the activity.

To evaluate the adapted code book, two authors coded 100 activities from randomly selected episodes, resulting in a Cohen's Kappa of 0.93, commonly regarded as almost perfect agreement.

Finally, to account for different tool usage base rates between programmers, we operationalized introspection tool

[2]The originally planned analysis for this experiment used an operationalization of tool usage based on metrics derived from automatically collected UI interaction events. These planned metrics proved to be too unreliable because of a high rate of noise in the interaction events. Thus, we decided to apply a post hoc analysis based on manual coding using the edit-run model.

usage as the ratio between the duration of inspect activities and the duration of all run activities (sum of inspect and test activities). We use the sum of run activities to detect shifts in how dynamic tools are used, independent of how the overall dynamic tool usage changes in relation to overall tool usage.

F. Experiment Procedure, Training Setups, and Balancing

We conducted the experiment via Zoom to reduce our presence during the task and to make it easier for participants to join, as they could more freely choose when and where they wanted to participate. Participants always worked in their own Zoom call. Whenever a participant worked on a task, we also deactivated both camera streams and microphones to give participants the impression of working completely on their own. We recorded the Zoom calls with the webcam streams hidden to keep the recordings anonymized.

While all participants had at least nine months of exposure to Squeak/Smalltalk, for some of them, the last encounter might have been more than a year ago. Thus, we asked all participants to join a two-hour training session on the days before their actual experiment run [38]. In the training session, we first recapped all relevant tools (dynamic and static) and keyboard shortcuts by demonstrating them. We then introduced the application, guiding them through a task, introduced a first task, and then let them work on example tasks in a training program for 1.5 hours. At the end of the training session, we asked participants to sign the participation agreement and reimbursed them (80€).

We started the four-hour main session by introducing the scenario through a short gameplay demonstration and a walk-through through the main packages. We then recapped the main tools and mentioned the potential delay, which we explained using a prepared text (see Section III-D). Afterwards, the main part of the experiment started.

The main run consisted of two parts. Each part was either under the live or the delayed condition. As we expected participants to require some time to adapt to the delayed and the live conditions, we began each part with a training task, which we did not analyze later on. Further, between the first and the second part, we introduced a mandatory 15-minute break, which we followed with another introduction of the main tools. Within each part, participants worked on two proper tasks, each with a time limit of $1\,\mathrm{hour}$.

We counterbalanced tasks by first splitting the four main tasks into two groups. Participants worked on each of these groups under one of the conditions. We combined task groups and conditions by using a balanced Latin square and then assigned the configurations to participants randomly.

For each task, we loaded the patch introducing the defect and then read the task description to participants. We asked participants to reproduce the failure while we still watched to ensure the failure was correctly reproducible in their environment. They then worked on the task and decided on their own when they completed the task by clicking on a button in the task management application.

After the two main parts, we asked them to fill out the demographics and experience questionnaire and answered any questions they had on the study.

IV. Tasks and Participants

As previous experiments on liveness indicated that *task complexity* and *participants' experience with liveness* change the impact of liveness, we controlled both of them [2], [3], [25].

A. Controlling Task Complexity

The tasks in this experiment were plain debugging tasks. Participants received a set of steps to reproduce some behavior, a description of the observable failure, and a target behavior.

The tasks were of similar structure to the tasks of a prior study [3] but otherwise entirely created from scratch and designed for a different game than in the prior study. The main challenge for designing these tasks was to control the task complexity [3], [39]. We needed to deliberately control the task complexity as we wanted to ensure that tasks are sufficiently complex to prompt the usage of introspection tools. Especially, as results of previous studies on liveness hint that programmers do not benefit from liveness when they work on simple tasks [2], [3], [24]. Further, because we chose a within-subject design, we needed to ensure that we kept task complexities comparable between tasks.

We designed moderately complex tasks while keeping the tasks solvable in the available time frame. For that, we used a framework for analyzing the task complexity[3] of tool studies [40]. The framework organizes task complexity factors along the variation points of tasks: task description, system (to be repaired), infection chain, patch, tool environment (with which defects are repaired), and general considerations.

To allow for observing potential introspection tool usage, the tasks should be designed so that programmers would benefit from using introspection tools. Thus, we aimed to make the tasks complex with regard to the infection chain and the patch. Correspondingly, we aimed to make the tasks in general simple with regard to the task description and the tool environment, as we want neither of them to interfere with tool usage. With regard to aspects of the system, we decided in more detail which should be simple or complex. We briefly outline the most important considerations in the following sections.

1) System: We chose a system that was simple in most regards but still allowed for complex enough debugging tasks. In general, to reduce repeated learning effects, we used the same system throughout the study.

We chose a game ("Realms of Zaltia", see Figure 3) as the system for this study, as participants were familiar with game development from university courses and thus would encounter fewer challenges with understanding the domain.

[3]We distinguish between task complexity, which is a property of the task itself, and task difficulty, which results from the combination of task performer and task.

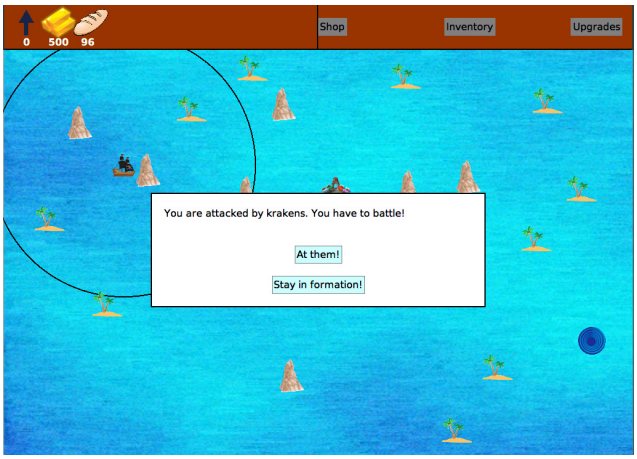

Fig. 3. A scene from Realms of Zaltia, a roguelike game similar to the game "FTL: Faster Than Light"[5]. Players command a ship of adventurers who navigate from island to island, trying to reach an exit point in time. Each island either holds a text adventure, a sea battle, or a fight on land. The game also features a shop system, equipment, and character leveling and attributes.

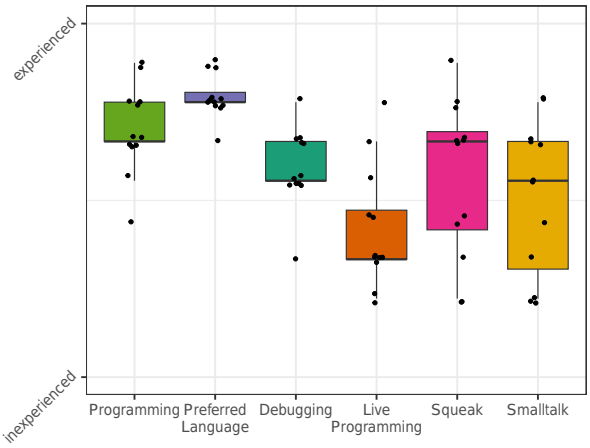

Fig. 4. Combined box and scatter plots detailing participants' self-reported experience. Each point corresponds to an answer by a participant. Participants could choose from a 10-point scale, with only the endpoints labeled.

Further, a game covers multiple kinds of dynamic behavior and distinct features, allowing for a more varied set of tasks.

The game was initially developed by university students in another edition of the same university course that our participants had taken. Consequently, the participants of our study did not know the game but would generally know how one would construct a game in the same programming environment. We repaired all remaining defects in the original game that may have interfered with our tasks to avoid ambiguous behavior of the system. Beyond this, we did not alter the game. Still, to reduce the complexity of the architecture, we guided participants through the package structure at the beginning of the experiment.

In general terms, the chosen game is relatively small in terms of LOC (see Table I), but this does not limit the complexity of the debugging tasks.

TABLE I
BASIC SOFTWARE METRICS OF THE GAME USED IN THE EXPERIMENT.

Metric		Metric	
#Packages	5	LOC	4611
#Classes	50	#Methods / Class	18.04
#Methods	902	LOC / Method	5.11

2) Task Description: We aimed to keep the task description as simple as possible. First of all, participants worked on one task at a time. We used a clear format, distinguishing between steps to reproduce a failure and the symptoms. All tasks used the same language throughout with the same formulations and terms[6].

[5]https://web.archive.org/web/20250329191555/https://www.mobygames.com/game/57826/ftl-faster-than-light/ (accessed 2025-05-12)

[6]For an example of this format used in another study, see [40, Fig. 2].

At the same time, we took care to give as few hints as possible on the infection chain beyond the failure.

3) Infection Chain: We aimed to make the infection chain complex enough to justify the usage of introspection tools, but also keep the tasks doable within the available time frame. To remove the complexity of deciding what to work on, each task only featured a single failure. In all tasks, the system only misbehaved and did not crash, as crashes are a clear entry point for the infection chain. Finally, the failures were easily reproducible.

4) Patch: To avoid ambiguity in the patch to be created, we designed tasks so that the target behavior is obvious when the infection chain is reconstructed successfully. Actually writing the patch should not be complex as it does not benefit much from introspection tool usage. It should be short and not require unusual classes or methods.

To remove the complexity of deciding when a defect is "good enough", we further told participants to work on the patch until they are as sure of it as they usually are when they commit to a repository of their own projects.

5) Tool Environment: We controlled the complexity of the tool environment by ensuring that all participants were aware of the same set of tools. For that, all participants took part in a tutorial beforehand and received recaps of the tools and shortcuts at the beginning and in the middle of the experiment (see Section III-F).

B. Participants

We recruited undergraduate and graduate university students ($N = 12$)[7] enrolled in the IT-Systems Engineering program. To only recruit participants who are experienced in working with liveness features, we required participants to have

[7]We recruited 17 participants in total, three participated in the pilot, two did not attempt the experiment main run. We have excluded the data of the two incomplete runs from all further analysis.

completed two undergraduate courses on Squeak/Smalltalk, resulting in at least 9 months of exposure to the environment. As we did not impose a limit on how long ago these courses could be, we controlled current skills with Squeak/Smalltalk through a training session before the experiment run.

The one female and eleven male participants were between 21 and 25 years old (median 22). We asked participants to self-report their experience levels in a questionnaire. Overall, they regarded themselves as experienced programmers, in particular in their preferred language (see Figure 4). They only felt somewhat experienced with live programming in general, but more so with the Squeak/Smalltalk environment and the Smalltalk language in particular.

V. Results

We gathered data from twelve valid experiment runs (see Figure 5) covering 1895 activities covering 18.5 hours of debugging time. Using this data, we tested the main hypothesis and examined the influence of task assignment on the results.

A. Analysis of Main Hypothesis

We tested the main hypothesis using a post hoc, one-sided, dependent t-test. The data met the assumptions of normality (Shapiro-Wilk test on the sample differences, $p = 0.495$), and the assumption of sphericity is automatically met as the independent variable only has two levels. The t-test results are significant at the 0.1 level, but not at the 0.05 level ($t(11) = 1.71, p = 0.058$). In particular, the usage frequency of introspection tools decreased from 0.91 ± 0.08 in the live condition to a frequency of 0.79 ± 0.20 in the delayed condition. The results suggest that removing liveness from introspection tools decreases their frequency of usage.

To cross-check that the trend does not result from a biased assignment of conditions to task sets, we also examined the distribution of condition assignments in the data (see color in scatter plot in Figure 5). Five participants worked under the assignment that assigned task set *A* to the live condition (*A-live*), and seven on the inverse assignment (*B-live*). The unbalanced number in the two groups is the result of two participants not completing the experiment. Still, we argue that this does not increase our chance of a type II error, as the assignment with the higher number of participants (*A-live*) has a wider spread (differences of feature usage per participant *A-live*: -0.259 to 0.559, *B-live*: -0.027 to 0.509, two participants used dynamic tools more in the delayed condition resulting in negative numbers) and a mean closer to zero (*A-live*: $\mu = 0.08, \sigma = 0.27$, *B-live*: $\mu = 0.18, \sigma = 0.21$). Thus, we argue that the task assignment only introduced a bias against our hypothesis, even though, as with previous experiments, the tasks seem to have an influence on the results.

B. Threats to Validity

We have identified two threats to the internal validity and two threats to the external validity.

The first threat to internal validity is that tasks differ in complexity and other characteristics. We argue that for this experiment, the influence of these differences was less severe than for other experiments, as we did not measure debugging efficiency. Still, as the comparison of the task assignments shows, task characteristics might influence how reduced costs affect usage frequencies of introspection tools.

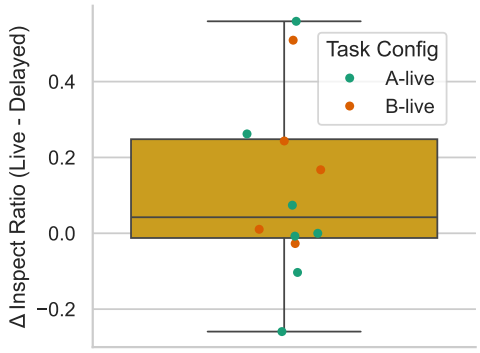

Fig. 5. A combined box and scatter plot showing the distribution of the differences in introspection tool usage for each participant. Each dot represents the difference between conditions for one participant. The color of the dots represents the assignment of task sets and conditions.

The second threat to the internal validity is the fact that in our experiment setup, we changed the properties of the programming tools that participants were used to. Thus, in the delayed condition, we disrupted workflows that they might have acquired for making use of liveness. Again, we argue that this is less severe as we did not measure debugging efficiency, but tool usage. Instead, we might have biased the results in favor of the delayed condition, as participants might have stuck to their workflows even though the delay became uncomfortable.

The first threat to external validity is the task design. We designed the tasks to be solvable in the given time frame and also to prompt the usage of introspection tools. Thereby, we might have selected a sweet spot where the cost reduction of liveness has a significant influence on programmer behavior. In practice, programmers might use introspection tools less often as many defects seem to be trivial enough to be repairable using only static tools [27], [28]. At the same time, when programmers use introspection tools in practice, the defects might be more complex than the ones we have used. Thus, for the common simple defects and the difficult defects, the value of the tools might be so low or high that the cost reduction from liveness does not move the cost-benefit ratio significantly.

Finally, participants were not allowed to change their programming tools and, through the artificial time limit, did not have the time to improve their workflow. Programmers experienced with the environment might have sought ways to reduce the costs of accessing dynamic information again, by improving their tools.

C. Discussion

The results of the post hoc analysis suggest that increasing the access time for introspection tools decreases their usage.

This finding is in line with our interpretation of liveness in terms of information foraging theory. By increasing the access time, we increased the actual cost of using these tools. Because the increase in access time was persistent, we argue that we also increased participants' estimates of the cost. As a result, the cost-benefit ratio for using introspection tools was lower than that of observing surface behavior, and participants shifted from introspection tools to only observing surface behavior.

At the same time, the results are not significant at the 0.05 level, and our analysis is a post hoc analysis. However, we still argue that the experiment offers valid evidence, as the effect is significant at the 0.1 level, the post hoc analysis has a straightforward interpretation, and the experiment setup aligns with the post hoc analysis, not least because the post hoc analysis is an adapted version of the pre-planned analysis.

Further, while these results support our proposed IFT interpretation of liveness, it is only a single experiment, and many other explanations might apply as well.

D. Observations from the Sessions

While the post hoc analysis offers a direct relation between tool access time and usage frequency, our subjective observations during the sessions offer some nuance to the result.

First, we expected participants to be used to short access times and perceive delayed actions as a nuisance. Interestingly, only one participant complained about the delays, even after we revealed to them that the delay was introduced intentionally. Several participants noted that they did not feel like the delays were a hindrance, as they were used to longer delays from their everyday work (e.g., one participant reported to do professional cyber-physical programming).

Similarly, we expected at least some participants to devise a new workflow that offers them a short access time, for instance by extensively logging state to the Transcript (the Smalltalk-equivalent of terminal output) and potentially devising short helper methods for doing so. However, only one participant used the Transcript extensively, and they actually also did so in the live condition, mostly to find suspicious state changes.

Finally, some previous works speculated that longer access times might lead to programmers adapting a more thoughtful workflow during which they make changes that are more considerate and access information more consciously. From our subjective impression, this did not occur during our sessions. Most participants continuously edited code or interacted with tools in small steps, independent of whether actions were delayed or not.

E. Future Work

While we argue that our IFT interpretation of introspection tools and liveness is more detailed than previous interpretations, it is still not fully developed. A more detailed interpretation for single tools (such as the debugger) or liveness features would allow for more detailed predictions. In particular, future iterations should include a more detailed definition of scent in introspection tools.

Similarly, our experiment only provides a first data point for exploring an IFT interpretation of liveness but is still too broad to allow for definitive conclusions, as it did not control for value and only changed cost in a general way. Valuable next experiments might focus on value or cost in greater detail.

For instance, to control the value of introspection tools more, participants might work with two tools with approximately similar features (e.g., the object inspector and explorer in Squeak/Smalltalk), with only one having an artificial delay added to accessing it. Another experiment might introduce a tool that always provides a very high value (for instance, might show relevant state infections that directly lead to the defect) and increase the cost of accessing it across conditions to determine whether participants are sensitive to cost or whether they will use the tool either way. We also expect the sensitivity to cost to vary with different kinds of information, e.g., programmers might be less sensitive to cost for dynamic information that is typically used in in-depth analysis such as detailed profiling information. In such cases, programmers might accept a higher cost and not change their usage patterns.

Similarly, we have only varied the temporal aspect of liveness in this experiment. A longer delay introduces obvious costs, but a larger distance between static and dynamic information and a higher conceptual distance could similarly be experienced as higher access costs. A fully formed IFT interpretation of liveness would consider these aspects as well.

Investigating existing design frameworks such as Tanimoto's liveness levels based on our IFT interpretation might also be fruitful. The first four liveness levels describe which parts of the process from static source code to dynamic information are automated and continuously updated. Each level, therefore, makes different kinds of information cheaper to access, and should therefore result in distinct workflow changes.

VI. Conclusion

To integrate liveness more closely with existing theoretical frameworks of programmer behavior, we proposed an interpretation of liveness in terms of the information foraging theory. For that, we introduced a detailed IFT interpretation of dynamic introspection tools. Within this interpretation, liveness reduces the cost of navigating to introspection tool patches. We hope that our IFT interpretation of liveness may inspire the discussion on the theoretical foundations of liveness and its role in programming tools.

To gather first insights on our interpretation, we conducted a post hoc analysis testing that an increase in access time to introspection tools increases their cost and, as a result, their usage frequency in the context of debugging tasks. The results are in line with our interpretation but are not sufficient to fully support or refute it. More detailed experiments are necessary to evaluate whether an IFT can model the impact of liveness.

References

[1] Rein, Ramson, Lincke, Hirschfeld, and Pape, "Exploratory and live, programming and coding - A literature study comparing perspectives on liveness," *The Art, Science, and Engineering of Programming*, vol. 3, no. 1, 7 2019.

[2] C. Cook, M. Burnett, and D. Boom, "A bug's eye view of immediate visual feedback in direct-manipulation programming systems," in *Proceedings of the Workshop on Empirical Studies on Programmers (ESP) 1997*, ser. ESP '97. New York, NY, USA: ACM, 1997, pp. 20–41.

[3] Rein, Ramson, Beckmann, and Hirschfeld, "Does task complexity moderate the benefits of liveness? - a controlled experiment," *The Art, Science, and Engineering of Programming*, vol. 9, no. 1, pp. 1–39, Oct. 2024.

[4] S. L. Tanimoto, "A perspective on the evolution of live programming," in *Proceedings of the Workshop on Live Programming (LIVE) 2013*, B. Burg, A. Kuhn, and C. Parnin, Eds. IEEE Computer Society, 2013, pp. 31–34.

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

[6] C. M. Hancock, "Real-time programming and the big ideas of computational literacy," Ph.D. dissertation, Massachusetts Institute of Technology, Sep. 2003.

[7] H. Lieberman and C. Fry, "Bridging the gulf between code and behavior in programming," in *Proceedings of the Conference on Human Factors in Computing Systems (CHI) 1995*. ACM/Addison-Wesley, 1995, pp. 480–486.

[8] D. M. Ungar, H. Lieberman, and C. Fry, "Debugging and the experience of immediacy," *Communications of the ACM*, vol. 40, no. 4, pp. 38–43, 1997.

[9] Pirolli and Card, "Information foraging." *Psychological review*, vol. 106, no. 4, p. 643, 1999.

[10] Fleming, Scaffidi, Piorkowski, Burnett, Bellamy, Lawrance, and Kwan, "An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 2, pp. 14:1–14:41, 3 2013.

[11] Lawrance, Bogart, Burnett, Bellamy, Rector, and Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2 2013.

[12] D. Piorkowski, "Information foraging theory as a unifying foundation for software engineering research : Connecting the dots," Ph.D. dissertation, Oregon State University, Sep. 2016.

[13] Alaboudi and LaToza, "Edit - run behavior in programming and debugging," in *Proceedings of VL/HCC 2021*. IEEE, 2021, pp. 1–10.

[14] A. Alaboudi and T. D. LaToza, "What constitutes debugging? an exploratory study of debugging episodes," *Empir. Softw. Eng.*, vol. 28, no. 5, p. 117, 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10352-5

[15] Lawrance, Bellamy, Burnett, and Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," in *Proceedings of the Conference on Human Factors in Computing Systems (CHI) 2008*. ACM, 2008, pp. 1323–1332.

[16] Pirolli and Fu, "SNIF-ACT: A model of information foraging on the world wide web," in *User Modeling 2003, 9th International Conference, UM 2003, Johnstown, PA, USA, June 22-26, 2003, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2702. Springer, 2003, pp. 45–54. [Online]. Available: https://doi.org/10.1007/3-540-44963-9_8

[17] A. Zeller, *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press, 2009.

[18] Perez and Abreu, "Cues for scent intensification in debugging," in *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013 - Supplemental Proceedings*. IEEE Computer Society, 2013, pp. 120–125. [Online]. Available: https://doi.org/10.1109/ISSREW.2013.6688890

[19] A. J. Ko and B. A. Myers, "Finding causes of program output with the java whyline," in *Proceedings of the Conference on Human Factors in Computing Systems (CHI) 2009*. ACM, 2009, pp. 1569–1578.

[20] S. McDirmid, "Usable live programming," in *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) 2013*, A. L. Hosking, P. T. Eugster, and R. Hirschfeld, Eds. ACM, 2013, pp. 53–62.

[21] D. Rauch, P. Rein, S. Ramson, J. Lincke, and R. Hirschfeld, "Babylonian-style programming - design and implementation of an integration of live examples into general-purpose source code," *The Art, Science, and Engineering of Programming*, vol. 3, no. 3, 2 2019.

[22] S. Lerner, "Projection boxes: On-the-fly reconfigurable visualization for live programming," in *Proceedings of the Conference on Human Factors in Computing Systems (CHI) 2020*. ACM, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3313831.3376494

[23] D. Ungar and R. B. Smith, "Self: The power of simplicity," in *Proceedings of OOPSLA 1987*. New York, New York, USA: ACM, 1987, pp. 227–242.

[24] Wilcox, Atwood, Burnett, Cadiz, and Cook, "Does continuous visual feedback aid debugging in direct-manipulation programming systems?" in *Proceedings of CHI 1997*. New York, NY, USA: ACM, 1997, pp. 258–265.

[25] Hundhausen and Brown, "An experimental study of the impact of visual semantic feedback on novice programming," *Journal of Visual Languages & Computing*, vol. 18, no. 6, pp. 537–559, 2007.

[26] Krämer, Kurz, Karrer, and Borchers, "How live coding affects developers' coding behavior," in *Proceedings of VL/HCC 2014*. Melbourne, VIC, Australia: IEEE Computer Society, 7 2014, pp. 5 – 8. [Online]. Available: https://doi.org/10.1109/VLHCC.2014.6883013

[27] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, "Studying the advancement in debugging practice of professional software developers," *Springer Software Quality Journal*, vol. 25, no. 1, pp. 83–110, 2017. [Online]. Available: https://doi.org/10.1007/s11219-015-9294-2

[28] M. Eisenstadt, "My hairiest bug war stories," *Communications of the ACM*, vol. 40, no. 4, pp. 30–37, 1997.

[29] C. Hundhausen and J. Brown, "What you see is what you code: A "live" algorithm development and visualization environment for novice learners," *Journal of Visual Languages & Computing*, vol. 18, no. 1, pp. 22 – 47, 2007. [Online]. Available: http://dx.doi.org/10.1016/j.jvlc.2006.03.002

[30] L. Gugerty and G. Olson, "Debugging by skilled and novice programmers," in *Proceedings of CHI 1986*, ser. CHI '86. New York, NY, USA: ACM, 1986, pp. 171–174. [Online]. Available: http://doi.acm.org/10.1145/22627.22367

[31] J. Hibschman and H. Zhang, "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, 2016, p. 233–245. [Online]. Available: https://doi.org/10.1145/2984511.2984570

[32] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 185–194. [Online]. Available: https://doi.org/10.1145/1806799.1806829

[33] J. Sillito, G. C. Murphy, and K. D. Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[34] L. J. Gugerty and G. M. Olson, "Debugging by skilled and novice programmers," in *Proceedings of the Conference on Human Factors in Computing Systems (CHI) 1986*. ACM, 1986, pp. 171–174.

[35] J. Kubelka, R. Robbes, and A. Bergel, "The road to live programming: Insights from the practice," in *Proceedings of the International Conference on Software Engineering (ICSE) 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 1090–1101.

[36] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. C. Kay, "Back to the future: The story of squeak - A usable smalltalk written in itself," in *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1997, Atlanta, Georgia, October 5-9, 1997*, M. E. S. Loomis, T. Bloom, and A. M. Berman, Eds. ACM, 1997, pp. 318–326.

[37] J. H. Maloney and R. B. Smith, "Directness and liveness in the morphic user interface construction environment," in *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology, UIST 1995, Pittsburgh, PA, USA, November 14-17, 1995*, G. G. Robertson, Ed. ACM, 1995, pp. 21–28.

[38] A. Ko, T. LaToza, and M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, 2015. [Online]. Available: https://doi.org/10.1007/s10664-013-9279-3

[39] D. G. Feitelson, "Considerations and pitfalls in controlled experiments on code comprehension," in *Proceedings of the International Conference on Program Comprehension (ICPC) 2021*. IEEE, 2021, pp. 106–117.

[40] Rein, Beckmann, Krebs, Mattis, and Hirschfeld, "Too simple? notions of task complexity used in maintenance-based studies of programming tools," in *Proceedings of the International Conference on Program Comprehension (ICPC) 2023*. IEEE, 2023, pp. 254–265.