# The Road to Live Programming: Insights From the Practice

Juraj Kubelka
Pleiad @ DCC, University of Chile
Santiago, Chile

Romain Robbes
SwSE @ Free University of Bozen-Bolzano
Bozen-Bolzano, Italy

Alexandre Bergel
Pleiad @ DCC, University of Chile
Santiago, Chile

## ABSTRACT

Live Programming environments allow programmers to get feedback instantly while changing software. Liveness is gaining attention among industrial and open-source communities; several IDEs offer high degrees of liveness. While several studies looked at how programmers work during software evolution tasks, none of them consider live environments. We conduct such a study based on an analysis of 17 programming sessions of practitioners using Pharo, a mature Live Programming environment. The study is complemented by a survey and subsequent analysis of 16 programming sessions in additional languages, *e.g.,* JavaScript. We document the approaches taken by developers during their work. We find that some liveness features are extensively used, and have an impact on the way developers navigate source code and objects in their work.

## CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**; **Object oriented development**;

## KEYWORDS

Live Programming, Software Evolution, Exploratory Study

**ACM Reference Format:**
Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2018. The Road to Live Programming: Insights From the Practice. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180200

## 1 INTRODUCTION

Live Programming aims to free programmers from the edit-compile-run loop of standard programming, going as far as to allow programs to be changed while they run. Recent years have shown that there is considerable interest in Live Programming, and liveness in general: the new programming language Swift from Apple features Interactive Playgrounds [1]; Microsoft recently purchased a company developing Live Programming tools [2]; Facebook's React supports instant feedback [3]; web browsers come with interactive consoles and inspectors [4]. Academically, a series of international conferences and workshops have been held on Live Programming, with LIVE 2013 being the most popular workshop at ICSE 2013 [5] (see Section 2 for background on Live Programming).

While some see Live Programming as "the future of programming", some of its proponents have more concrete and immediate issues. Microsoft Research's Sean McDirmid [6] wrote: "The biggest challenge instead is live programming's usefulness: how can its feedback significantly improve programmer performance?"

While the full vision of Live Programming is yet to be implemented and made available at scale to practitioners, many elements of it are in use on a daily basis in some developer communities. In particular, many of the Live Programming concepts have been present for decades in the Smalltalk and Lisp development environments; Smalltalk and Lisp feature high degrees of *liveness*.

This offers us a critical opportunity to understand *how* liveness is used *today*, and offers insights to practitioners, researchers, language designers, and tool builders on how to effectively use and support Live Programming. In this paper, we provide answers to several research questions on the use of Live Programming features in practice. We first provide essential background on Live Programming and our main study subject, the Pharo IDE, in Section 2.

We then present our methodology (Section 3): we analyzed 17 development sessions of 11 programmers, totaling 13 hours of coding with familiar and unfamiliar code. This was followed by two confirmatory phases: a survey of 190 Smalltalk developers, and an analysis of 16 online videos (25 hours) of HTML/JavaScript, C#, PHP, Haskell, and C/C++ programmers.

Section 4 answers our first research question, **"Do developers use the Live Programming features of Pharo?"**, in the the affirmative: we find that Live Programming features and tools were used extensively by Pharo users.

Section 5 answers our second research question: **"How do developers use Live Programming in Pharo?"**. We describe several usages of Live Programming and contrast them against traditional programming approaches. Our overall finding is that simple approaches were favored, and their combinations can be powerful.

We then answer our third research question **"Do developers, other than those we analyzed, behave similarly?"** in two parts. First, we find that surveyed Pharo users agree with most of the observations we made (Section 6). Subsequently, we observe that users of other languages and tools use more limited capabilities due to the limited extent of the tool support, and with the aim of getting feedback early and often (Section 7).

We then discuss our findings in the context of existing studies and tool support in Section 8. In particular, we discuss how a reduced toolset, centering on the concept of the *object inspector* supports a large range of use cases.

Finally, we close the paper with a discussion of the threats to validity of this study in Section 9 and conclude with implications for tool builders, researchers, and language designers in Section 10.


*ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180200

# 2 BACKGROUND AND RELATED WORK

## 2.1 Live Programming Environments

The reader wishing to experience Live Programming concretely may visit the http://livecoder.net web site for a JavaScript example.

*Degrees of liveness.* Live Programming allows programmers to obtain nearly instantaneous feedback from their programs through always accessible evaluation [7]. Tanimoto describes four levels of liveness [8]. At level 1, the user must restart the whole application to obtain any kind of feedback. At level 2, semantic feedback is available on-demand; this is the level provided by interactive interpreters (Read-Eval-Print loops). At level 3, incremental feedback is automatically provided after each edit operation. Finally, level 4 applies changes to a running application without explicitly initiating the application under change. Each level creates a different experience and cover solutions such as auto-testing, Read-Eval-Print loops (REPLs), or systems where development tools and applications share the same environment [9].

While elements of Live Programming date back decades, with some versions of it supported in LISP's Read-Eval-Print Loop [10], or in Smalltalk and its descendants (*e.g.,* Self [11], Squeak/Smalltalk [12] or Lively Kernel [13]), it has seen a resurgence of interest in academia and in the industry.

*Live Programming research.* Academically, both the LIVE programming workshop [14] and the International Conference on Live Coding [15] held their 3rd edition in the fall of 2017. Many Live Programming languages and tools have been proposed, but most of them are experimental. As we focus on the usage of liveness in practice—and due to space constraints—we only give a brief coverage of a handful of seminal work. The 90's saw an interest in Live Programming for visual languages, such as Tanimoto's VIVA [16], that introduced the 4 liveness levels. Burnett *et al.* implemented level 4 liveness in a visual language [17]. They also conducted a controlled experiment of debugging comparing live and non-live tasks [18]. They found that subjects performed twice as many changes in live tasks, but the effect in terms of time and accuracy was mixed. Recent works include languages such as McDirmid's SuperGlue [19], Jonathan Edward's subtext [20], and Glitch [21]. Microsoft's TouchDevelop has been modified to support Live Programming [22]. DeLine *et al.* proposed Tempe, a Live Programming environment for data analysis [23].

*Liveness in practice.* Several frameworks and tools nowadays add ways to integrate immediate feedback, particularly on the web. The Chrome web development tools [4] allow changes to web pages and JavaScript code without reloading a webpage. Modern, popular JavaScript frameworks including Facebook's React.js support Live Programming in live code editors [3].

Mainstream languages also take steps in this direction: Java supports a limited replacement of application parts while it is running through the Java Platform Debugger Architecture [24]. Apple's new language Swift supports Live Programming with interactive playgrounds [1]. Finally, Microsoft's Visual Studio 2015 supports interactive programming for C# and F# with REPLs [25, 26]. Microsoft also recently acquired the CodeConnect startup, which developed the Alive extension for Visual Studio [2].

## 2.2 Live Programming in Pharo

*Why Pharo?* Pharo [27] is a programming language and development environment derived from the Squeak/Smalltalk [12] dialect of Smalltalk-80 [28]. As the development tools and the applications share the same runtime environment, developers naturally work on running applications. In addition, developers constantly improve Pharo itself by altering it while it is running. Since Pharo can update any part of an application at runtime, it supports up to Tanimoto's fourth level of liveness. As a descendant of Squeak, Pharo also features the direct manipulation Morphic UI [29], originally implemented in Self [11].

As a descendant of Smalltalk-80, liveness is present in Pharo since more than three decades. This has two consequences that made Pharo the case study of choice in this work. First, liveness was designed in the language and the tools since its inception; the support is both mature and seamless. More importantly, liveness is also part of the culture of the development community. Pharo is used by an active community of developers, both in industry and academia. Pharo users are introduced to liveness when they learn the language from the very beginning, and as a result practitioners use it naturally as part of their daily tasks. This characteristic is the main driver of our choice of Pharo, as we wanted to know how developers familiar with liveness use it, in practice.

*Liveness in Pharo.* We now present the main liveness features of Pharo, contrasting it with a traditional IDE, Eclipse (assuming a default configuration).

*Code evaluation.* The first difference is the ease of evaluating code in Pharo. Since the compiler is a user-accessible object, *any piece of text* can be selected, and then be executed, printed out, inspected, debugged, or profiled, either via menus or keyboard shortcuts. Eclipse can only run applications in normal or debug mode, and execute tests. This difference has far-reaching consequences. For instance, in Pharo it is common to encounter executable examples embedded in method comments. To better understand a method, a user can simply select the example, execute it, and observe the result.

Pharo can also compile and update a single method at a time. This can be done even while applications are running: any new call to the changed method from then on will call the updated behavior.

*Accessing objects.* The Object Inspector is a tool that displays object states [30]. Developers can navigate the object graph, modify instance variables, evaluate code snippets, browse its source code, and invoke inspectors from virtually anywhere. They may open multiple inspectors at the same time and keep them for arbitrary long periods of time. This is useful for contrasting the object behaviors and for evaluating the result of source code change on live objects. The inspector can be embedded in other tools [31]. Many objects have custom inspector views.

Eclipse has a more limited Object Inspector in its debugging perspective that can only navigate the object's structure, change the values of primitive types, and is dismissed as soon as the developer leaves the debugging perspective.

*Playground.* The Playground is a tool to write code snippets and explore their results, which are displayed in an embedded Object Inspector. Playgrounds can be used to experiment with example code snippets. Developers can open various playgrounds and keep them open indefinitely. Eclipse does *not* provide a similar tool.

*Debugger.* The Pharo Debugger is composed of a stack trace at the top, a source code editor in the middle, and an embedded inspector in the lower part with instance and method variables. Developers can change the variable values, edit source code, and observe the effects of these changes. Pharo developers can have several debuggers open at the same time, which allows for easy comparison of the execution of two execution scenarios. When a non-existent method is invoked on an object, the debugger offers to create it, generates a stub, and then navigates inside the method stub, allowing the developer to implement it with live arguments and variables at his or her disposal. This process, called "programming in the debugger" can be repeated indefinitely (*e.g.,* a newly added method can in turn call a non-existing method), and is especially popular among proponent of test-driven development.

Eclipse developers can evaluate code snippets inside of a debugger. However, Eclipse does *not* support multiple opened debuggers. Changing source code at run-time is limited to methods visible in the call stack; any other change needs an application restart. Pharo developers can change any source code elements, including class definitions. The Pharo debugger is automatically invoked whenever an error occurs, while Eclipse's has to be invoked manually.

*Halo.* Originating from Morphic [32], the Halo enables interactions with on-screen widgets. The Halo is invoked on the UI element under the mouse via keyboard shortcuts. Buttons appear around the widget, triggering actions such as copying, altering, deleting, or inspecting it. Eclipse does *not* offer a similar functionality.

## 2.3 Studies of Developers

Many observational studies of developers have been conducted. We very briefly list some here—see Section 8 for discussions in context.

Ko *et al.* studied how developers seek information during maintenance tasks [33]. Lawrance *et al.* studied how developers debug from an information foraging perspective [34]. LaToza *et al.* present developer activities and the tool support level for them [35]. Roehm *et al.* conducted an observational study about software comprehension in seven companies and report on comprehension strategies [36]. LaToza *et al.* compared novice and experts during program comprehension [37]. Robillard *et al.* contrasted 5 successful and non-successful developers during maintenance tasks [38].

As they work, developers ask many questions. Several studies seek to catalogue them and analyze their tool support. Sillito *et al.* [39, 40] observed 27 software maintenance sessions, and listed 44 questions asked during them. LaToza *et al.* focus on reachability questions and provide 12 questions rated by difficulty and frequency [41]. Ko *et al.* identify 21 questions about interaction with codebases and co-workers [42]. Fritz *et al.* present 78 developer questions with lack of tool support that involve information integration from various sources [43]. Duala-Ekoko *et al.* conducted a study on unfamiliar APIs and identified 20 questions [44].

Some studies disallowed the use of runtime information (*e.g.,* Robillard *et al.* [38]); others did allow it, but did not specifically report it (such as Ko *et al.* and Lawrance *et al.* [33, 34]). Thus, studies of live environments, where a high amount of runtime information is available, complement previous work. DeLine and Fisher performed such a study, but in the restricted context of Data Scientists [45], not in software development.

**Table 1: Participant information.**

| Participant Id | Programming experience in Any language [years] | Smalltalk [years] | Current Position | Conducted Sessions |
|---|---|---|---|---|
| P1 | 5.5 | 3 | Ph.D. Student | S1 |
| P2 | 15 | 11 | Professor | S2 |
| P3 | 5 | 1.5 | Professional | S3 |
| P4 | 20 | 13 | Professor, Professional | S8, S11 |
| P5 | 7 | 0.5 | Master Student, Professional | S9, S12 |
| P6 | 22 | 16 | Professor | S10, S13 |
| P7 | 10 | 6 | Ph.D. Student | S4 |
| P8 | 4 | 2 | Bachelor Student | S5, S14 |
| P9 | 7 | 3 | Ph.D. Student | S6, S15 |
| P10 | 3 | 0.5 | Professional | S16 |
| P11 | 5 | 3 | Ph.D. Student | S7, S17 |

# 3 RESEARCH METHOD

## 3.1 Exploratory Study

*Participants.* We recruited eleven male participants consisting of students, academic staff, and professional developers from distinct small local companies. Their programming experience spanned from 5 to 22 years, with a median of 6 years. Experience in Pharo or Smalltalk ranged from 0.5 to 16 years, with a median of 3 years. The participants consisted of 1 bachelor student, 4 Ph.D. students, 2 professors, 2 professional developers, and 2 persons with mixed positions. Details are illustrated in Table 1.

*Tasks.* We conducted 17 programming sessions, of which 10 sessions (S1–S10) included an *unfamiliar codebase*, and 7 sessions (S11–S17) a *familiar codebase.* A session was classified as *familiar* if the participant was one of the authors of the codebase. In this case we asked the participant to choose a programming task of his or her project on which he or she can work during 40 minutes, and stressed it was not important to finish the task at the end of the session. In *unfamiliar* sessions, the participant had little or no knowledge of the codebase beforehand. We provided him or her a prepared Pharo image related to a particular bug-fixing or extension task and an explanation of the task.

*Study Setting.* Participants completed the study using Pharo version 3. They could use any IDE features, and any documentation resources. They were advised to proceed with their work as usual. Before each session, we setup the participant's device and explained the procedure of the session. We asked participants to verbalize their thoughts while solving their tasks. Participants did not receive any additional prior training for our study. After about 40 minutes, we informed participants that they could finish whenever they wanted.

*Data Collection and Transcription.* We used three data collection techniques: screen captured videos (including audio, 13 hours in total), user interactions, and interviews. We summarized what the participants said, and described the actions they were performing on the screen; we developed tools to attach timestamps to the transcripts and navigate back to the videos as needed during the analysis. These data, recordings, and tools are available at [46**? ?** ].

We complemented this data with IDE user interactions [47], which we captured with the DFlow framework [48]. Events included opening, closing, and moving of windows, clicks on UI elements, navigation and changes to the source code, *etc.* (see details in [48]).

The interactions were not recorded in the session S14 due to the participant's incompatible setup; we exclude S14 when we report on DFlow data (in Section 4, and in Figure 2; we manually checked the presence of each tool but did not count the windows).

After each session we conducted semi-structured interviews in which participants were asked to comment on the challenges experienced during the sessions and divergences with their day-to-day work. The interviews lasted from five to fifteen minutes.

### 3.2 On-line Survey

Since our aim was to confirm the exploratory study findings, we used an on-line survey as it is a data collection approach that scales well [49]. The survey is split into three parts: demographic information, tool usage questions with multiple choices, and open-ended questions. None of the questions were mandatory. Excluding demographics questions, the survey consisted of 34 multiple-choice questions and 2 open-ended questions, and answering it took about 10 minutes. The survey is available at [46].

We first ran a pilot survey with a limited number of people to clarify our questions. We then advertised the survey through seven mailing lists related to Smalltalk communities, and Twitter. To attract participants we included a raffle of two books.

The survey received 190 responses. Our participants were experienced (60% more than 10 years; 10% 6 to 10 years; 16% 3 to 5 years; and 14% 2 years or less), and many were from industry (44% from industry, 19% from academia, 24% both, 13% others, *e.g.,* hobby). This balances with our study participants, primarily from academia. Respondents used several Smalltalk dialects; 70% programmed daily.

### 3.3 On-line Coding Session Videos

As conducting an exploratory study is a time-consuming process even with a limited number of sessions, we complement this study with an analysis of publicly available coding session videos. We searched for programming videos on YouTube, using keywords such as "coding session", "programming session", and "live programming", and environments and frameworks that were more likely to feature a degree of liveness, such as "Javascript", "React", "C#", or "Swift". We also considered the related videos proposed by YouTube. We selected potentially relevant videos based on their title and skimming their contents. The initial list included 44 videos, totaling about 50 hours. This data is available at [46].

In the second step, we watched and categorized each video in order to determine whether it is a programming session or a pre-arranged tutorial or talk. We end up with 16 coding session videos of about 25 hours. We then carefully watched these videos, looking for five events: (1) a developer observes a running application or variable values after a change; (2) a developer modifies source code or variable values of a running application; (3) a developer uses an inspector to get runtime values; (4) a developer uses a debugger to understand a problem; (5) a developer write and run code snippets. For each event we identified whether: (i) we observe it at least once; (ii) we observe a part where it is used iteratively after a small meaningful change; or (iii) we do not observe it.

The video subjects use the following programming languages: 11 HTML/JavaScript (69%, 15 hours), 2 Haskell (13%, 3 hours), 1 PHP (6%, 2 hours), 1 C# (6%, 4 hours), and 1 C/C++ (6%, 14 minutes).

## 4 RESULTS: USAGE OF LIVENESS

In this section, we are concerned whether participants of the exploratory study use the Live Programming features. To answer it, we divide development tools into two categories: *static tools* if the main purpose is to present source code (*e.g.,* Source Code and References Browsers); and *dynamic tools* if the main purpose is to present the state of an application (*e.g.,* Debugger, Playground, Inspector, Test Runner, Profiler). The goal of this classification is to get us a first-order approximation of the use of Live Programming. This is only an approximation, as developers can change source code with static tools and have an immediate feedback, or use dynamic tools and observe application states without performing changes.

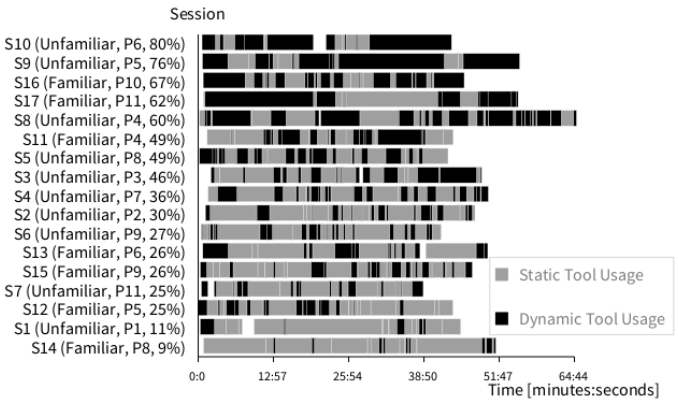


**Figure 1: Dynamic and static tool usage per session.**

Figure 1 shows the usage of static (gray) and dynamic (black) tools over time in all the sessions. Each session label includes the following information: session ID, unfamiliar or familiar task, participant ID, and dynamic tool usage percentage. In total, our participants spent 5.5 hours (42%) using dynamic tools and 7.5 hours (58%) using static tools, with 5 sessions having 60% or more of the time spent with dynamic tools. In addition, we can see that developers routinely switched between static and dynamic tools, even for short durations. Those numbers highlight the high overall usage of dynamic tools during the sessions, and its frequency.

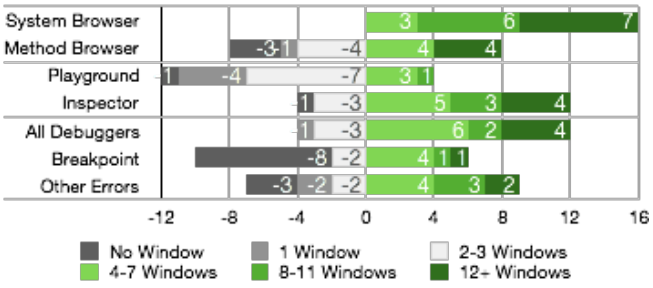


**Figure 2: Number of windows opened per session**

Figure 2 shows, for the most frequently used tools, the number of windows opened in each session. Results are aggregated for

compactness: each tool is summarized by a bar divided in cells. Each cell represents the number of sessions for which a specified range of windows were opened; gray cells to the left indicate sparse or no usage of a tool, while green cells to the right indicate prolific users. We subdivide debugger usage in two categories: explicit user invocation with breakpoints, and automatic invocation by the IDE when runtime errors occur.

Figure 2 allows us to easily rank tools by usage. The system browser (222 of 750 development tool windows, or 30%), used to browse and edit code, unsurprisingly goes first: all subjects used it extensively. Then comes the object inspector: all but one participant used it (139 total windows, 19%), and many did very frequently. Our most prolific participant opened 30 of them! Note that this is an underestimate, as this number does not include inspectors embedded in other tools (debuggers, playgrounds). Debuggers caused by runtime errors follow (91 windows, 12%). They were present in all but 3 sessions, and appeared for the following reasons: non-existent methods (60), test case assertion failures (19), and other runtime errors (12). The method browser used for reference navigation comes next (91 windows, 12%), being extensively used by half of the participants. Next, the playground appears (46 windows, 6%), although this metric may be somewhat misleading: all but one participant used playgrounds, and these windows tend to be long-lived. Finally, debuggers invoked on breakpoints are next (47 windows, 6%), with 8 participants never using them. Other tools such as test runners (2 windows), profilers (3 windows), or version control tools (16 windows) saw little use.

*Summary: Participants used dynamic tools frequently. Chief among them was the Object Inspector.*

## 5 RESULTS: LIVENESS EPISODES

In this section, we describe the strategies that programmers took to make progress in their tasks. They took a variety of approaches, sometimes taking advantage of Pharo's liveness (accessing objects, finding and modifying examples, crafting code snippets, modifying applications), sometimes employing more traditional approaches (unit testing, debugging, code search, and static exploration). Each type of usage (or *episode*, reusing the terminology of Zieris and Prechelt [50]) is illustrated by one or more examples from the transcripts. Finally, we highlight how the approaches can be combined.

### 5.1 Accessing and Manipulating Objects

Accessing objects with the inspector is the most common strategy participants took; it was present in all but one sessions. Inspectors were used frequently due to Pharo's ease of executing source code anywhere, including in code browsers, playgrounds, debuggers, and inspectors themselves. Usages vary in sophistication from simple accesses (checking of class names, variable values), to manipulation through execution of method calls and code snippets on the objects.

*Basic usages.* Participant P4 (S8) copy-pasted an example in a playground in order to understand a library. He had no idea what a variable was; its name "`a`" was far from useful. By adding the method call "`a inspect`" at the end of the example and executing it, P4 received an inspector on "`a`", where he could see the object's state, including its class name.

*Using multiple inspectors.* An important feature of the inspector is that one can keep any number of inspected objects opened, for as long as necessary. In the case of Eclipse, developers lose access to objects when they exit a debugging session. Pharo developers thus can change code and regularly execute methods in previously inspected objects to see the impact of their changes.

Participant P9 (S6) worked with the Rubric framework that creates text editors. P9 found out how the library works by opening inspectors on objects of interest: P9 obtained an inspector on a UI widget representing an editor view. The view had a model holding information about the contained text. Within the inspector, P9 evaluated and asked for the view's model and opened it in another inspector. Then P9 was sending messages to both objects, and observed how they interact and change each other.

*Inspecting UI objects.* Participant P11 (S7) wanted to find where a mouse click that opened a UI window is handled. He inspected the UI window using a halo (see Section 2.2), and found the window's class. He then put a breakpoint in the class constructor, triggered the mouse click again, and found the source code in the debugger.

Note that being able to inspect graphical widgets does not always result in finding relevant information. Similarly to P11, Participant P5 (S9) wanted to know where a particular dialog was executed. He inspected the dialog widget, but started diving in the (complex) structure of the object, expecting to find a relevant method name (this approach is possible, but is challenging). P5 spent about 20 minutes before realizing he could do like P11 who subdivided the question in two steps: identifying the widget class, then using a breakpoint to find out where it is created.

Using multiple inspectors with halos is possible too: Participant P6 (S10) had to fix a bug relating to an application's text editing area. Noticing that the bug is not present in another application, he used halos to find out whether the text editing areas in the two applications were similar; they were of two different UI widget classes. P6 could then focus on one class of widgets only.

### 5.2 Finding, Modifying, and Executing Examples

Our participants found examples from a variety of sources: in documentation, in source code, in test cases, and in dedicated browsers. They were easy to execute and change, but more complex examples were more challenging to reuse. Even if seldom used, we observe that having an example browser with running (living) examples simplifies exploration and comprehension.

*Basic usage.* After identifying the class of the text area (see above), P6 searched for references of the text area class. Executing some of the examples he found in his search, he concluded that the problem is not in the widget itself, but likely in its configuration or usage.

*Example browser.* Participant P4 (S8) was tasked with enhancing the Glamour UI framework: P4 had to show that time-consuming UI operations were in progress, avoiding the impression that the UI is frozen. P4 first searched for a code example that had such a time-consuming operation in the Glamour example browser. The example browser is a browser that only shows methods annotated as examples; the examples are automatically executed, showing the results with a domain-specific view (for Glamour, it shows the UI generated by the example). When he found a suitable example, he

pasted it into a new playground to execute it. Once he was familiar with it, he made changes related to his task.

*Isolation issues.* Participant P9 (S6) had to replace the legacy text editor framework in the Nautilus browser with a new framework, called Rubric. He found an example of a Rubric editor. Despite having a working example, it took P9 about two and half minutes to isolate the relevant parts in order to reduce the size of the example to the minimum working solution.

## 5.3 Crafting Code Snippets

Pharo's playground is a simple but powerful combination of a code editor and an inspector; participants used it to write small code snippets that helped them evaluate solutions and isolate bugs.

*Isolating bugs.* Participant P11 (S17) explored performance issues in an application. He thought that the issue may be caused by one of three libraries—three application layers. To remove suspicion from the uppermost layer of the application, he wrote a code snippet that was a minimal prototype of the application and only used the second layer. As the owner of the second layer library, he wrote it from memory in about two minutes. After four minutes, he had a working prototype and discarded the third layer from suspicion.

*Multiple snippets.* Participant P4 (S8), tasked with enhancing Glamour's responsiveness (see above), found a non-responsive example to get started. P4 constructed possible solutions, writing prototypes in different playgrounds. Thus he thought up of several solutions to the task, made changes in the library, and observed the behavior of the running prototypes. P4 said that he usually keeps examples, because "*it is an easy way to get back [to an earlier version] or share gained knowledge with others.*"

## 5.4 Modifying Running Applications

Participants frequently manipulated running applications, particularly when a feature is accessible from the GUI.

*Immediate feedback.* Nautilus is a source code browser in Pharo. P11 (S17) was asked to make its fast navigation functionality user-visible. P11 worked in Nautilus itself, iteratively performed changes in the codebase, and immediately tested the change effects in the running application. This way, he verified the correctness of his progress. In this case, it was not necessary to recompile and restart Nautilus each time he made progress.

*Misleading feedback.* Participant P9 (S6, see above) was also asked to change Nautilus: P9 had to replace an old text editor framework with a new one called Rubric. Upon finding relevant source code, P9 modified it to ensure that he found the right location. However, his changes were unexpectedly *not* visible in his running version of Nautilus. P9 spent about three minutes understanding why the changes were not reflected in the running application: the method P9 modified is called only once during the startup of Nautilus (this could be seen as a limitation of Pharo's liveness). P9 opened a new browser and was finally able to see the changes.

## 5.5 Traditional Approaches

We briefly cover approaches documented in other work.

*Test Cases.* We found overall few uses of test cases; they were more common on familiar codebase. Participants P4 (S11) and P8 (S14) used the test driven development (TDD) methodology: They first wrote test cases, then enhanced source code to satisfy the tests. The tests ensured their solution's accuracy. Both were working on familiar code, and knew since the start what they had to do.

P4 stated during his first session (S11) that he always uses TDD, but he did not write any test cases in session S8, on unfamiliar code. He confirmed that in this case, he did not have enough knowledge to write test cases. P4 instead wrote several prototypes that reflected his ideas, and then iteratively made changes to the code that he tested on his prototypes. Similarly, participant P6 (S13) was enhancing a graphical user interface. He expressed—while repeatedly manually testing the user interface—that he usually writes test cases, but "*it is hard to write test cases for graphical user interfaces.*"

*Using the debugger.* Participants used a debugger at least once in each session. Only half of them used breakpoints; the others only reacted to errors. Participants did not usually change source code in the debugger. Typically, the errors had to be fixed in source code locations other than where the error occurred. P4 (S11) fixed a few test cases directly in the debugger, when it was possible to do changes locally. We did not see instances of "programming in the debugger".

When using breakpoints, participants were usually interested in variable values and where a method is called from. This helped them to comprehend programs (*e.g.,* P11 in S7 above). P8 (S5) used the debugger to find where and how a particular application part is initialized. Later, he regularly changed the code, executed it, and watched if the implemented parts worked and what was necessary to fix next. He rewrote his code several times. One reason was that the objects he needed were available in different moments of the execution. He gained the knowledge during his work.

P6 (S10) spent nearly the entire session in a single debugger, exploring in detail a long method with several conditionals. He carefully read each statement, evaluated expressions to check their return values, and observed all variable values. Finally, he identified where to do the change, and changed it (in the debugger itself).

*Text Search.* Along with reading source code statically, search is a common tool in traditional IDEs. We observed few text searches. A possible reason is that Pharo has fragmented search tools, that increase the cost of searching: one tool searches in source code text, while others do semantic searches *e.g.,* class and method names.

Participant P5 (S12) looked for a class implementing a tree graph, searching for classes including the word *tree*. He was not successful, but was convinced that "*something as fundamental as a tree object structure has to exist*". He thus searched the Internet and an online book. He did not find anything and solved his task differently.

P6 (S13) explored a class and looked for a method using auto-completion, saying, "*there should be a method named something like open.*" After several tries he found the method `openWithSpec`.

*Source code exploration.* When all else failed, participants had to resort to static source code exploration. Deciding when to stop reading code and start changing it was not trivial.

Stopping too early, one risks missing important information and make incorrect decisions. P2 (S2) was on the right track at the beginning of his session. Before any changes, he first statically explored code in order to confirm his thoughts. He focused on the right code, but dismissed details because he thought it was "*too low-level code.*" He could not confirm his assumptions, missed important code for his task, and ended up with a complex solution.

Neither did exploring more than necessary lead to good decisions. P1 (S1) spent his session mostly browsing code. He soon ended up in a framework that was not important to understand; he spent almost half his time exploring it. While he gained a comprehensive knowledge about it, he did not find a starting point for the task.

### 5.6 Combining Approaches

These episodes are not observed in isolation. Participants switched techniques during the course of a session as necessary, and also combined techniques, thanks to Pharo's ability to open unlimited instances of each tool. For instance, P11 combined halos, inspectors and debugging in S7 (above). This is best exemplified by Participant P7, while seeking to isolate a bug in a graphical library:

P7 (S4) wrote a code snippet in a playground. The code represented the minimal steps to reproduce the bug in the application. In addition to running the application, P7 was concerned about a particular object it used. He thus added the `inspect` message to his snippet and executed it, obtaining the application's graphical window and an inspector with the object of interest. He then crafted a similar code snippet that involved other UI widgets, and behaved correctly. He executed this second snippet, and opened a second inspector. P7 regularly modified and executed the code snippets on both cases and observed the differences in behavior. By observing both running examples and the corresponding source code, he was able to reveal the problem and fix it.

*Summary: Participants employed a variety of approaches, but preferred simpler ones. Combining approaches had powerful results.*

## 6 RESULTS: ON-LINE SURVEY

In this section, we present the survey results. Figure 3 displays an overview of frequency answers, Figure 4 then reveals agreement responses. Due to the space constraints, we simplified the wording of the individual claims. For easier comparison of the claims, the responses "Rarely", "Occasionally/ Sometimes", "Regularly", "Disagree", "Somewhat disagree", and "Neutral" are aligned on the left.

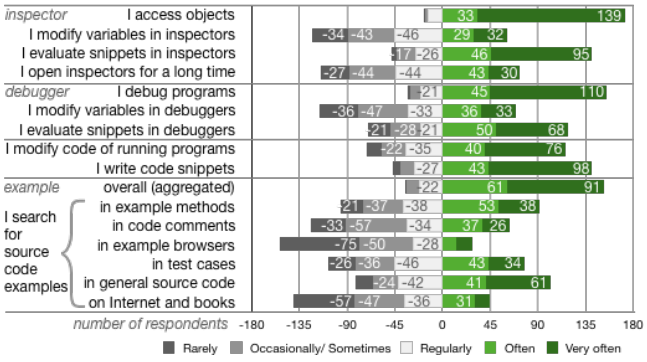


**Figure 3: Frequency answers.**

*Tool usage.* We find that our respondents indicate that they use the liveness in Smalltalk IDEs overwhelmingly and often. All but 3 of the respondents (98%) indicated that they used at least one of the tools often. The inspector once again took the lead (172 respondents used it at least often, 90%), followed by the debugger (all debugger usages, 155 respondents, 82%), then examples (aggregated, 152 respondents, 80%), code snippets writing tools (*e.g.,* Pharo's playground, 141 respondents, 74%). Modifying running applications was somewhat less popular (116 respondents, 61%).

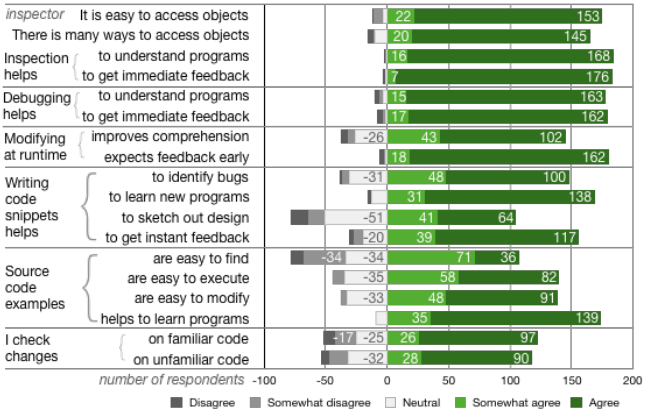


**Figure 4: Agreement answers.**

*Specific practices.* We note that keeping inspectors open for long periods of time (more than one hour) is practiced often by a respectable amount of our respondents (73 respondents, 38%), and regularly by an additional 44 respondents (23%). Participants also tended to evaluate code snippets more than directly changing data structures, be it in the inspector or the debugger (74 vs 32%, and 62% vs 36%). This also indicates that the playground is not the only place where code snippets are crafted. We note that examples were popular overall, but that the sources are varied. Dedicated example browsing tools, still relatively new, were not used extensively.

*Agreement answers.* We find that our respondents generally agreed with the statements we proposed. There is a particularly broad agreement with our assertions on the inspector, and the debugger. Agreement is more moderate in the case of examples (finding relevant source code examples is not always easy), and code snippets (code snippets are less used to sketch out program structures). Respondents found that fast feedback was somewhat easier to achieve in inspectors and debuggers than with code snippets (presumably since many queries in the inspector or debugger do not involve writing code). They finally indicated a strong preference to check their changes as often as possible, and not only after finishing a task. We restate that all of the statements proposed were generally met with agreement.

*Comparisons.* We also asked our participants to compare reading code with other approaches, on both familiar and unfamiliar code. We briefly relate the results for space reasons. Participants indicated that they found it easier to read code than to write either tests or code snippets. On the other hand, they indicated that they found it easier to understand programs by running them (or running examples), rather than reading code. In all cases, reading code was slightly easier when said code was familiar.

*Summary: Survey participants confirmed that they used liveness extensively; the inspector was again the most frequently used.*

## 7 RESULTS: ON-LINE CODING SESSIONS

In this section, we present the results of on-line coding session videos. Figure 5 displays our observations for each programming approach. For easier comparison, the conclusions "Not Observed" have negative values.

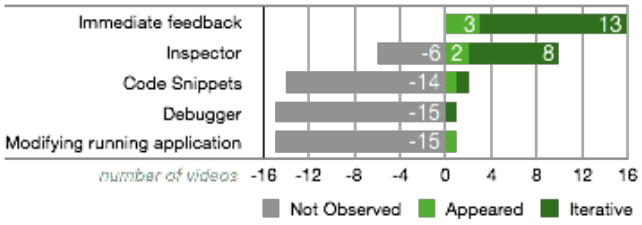


**Figure 5: On-line coding session video observations.**

*Immediate feedback.* This was the most used approach. Participants usually tried to write the minimum amount of source code that they were able to run and test; sometimes an incorrect behavior was expected. In some cases, the application restart was performed automatically, others manually. Two programmers used test cases that were re-executed automatically in a text console.

*Inspector usage.* An object inspector, in particular HTML/DOM inspector and JavaScript object inspector, was the second most frequent tool. Developers predominantly inspected HTML pages and explored HTML DOM and CSS rules. JavaScript's object inspector was frequently used in a browser console in order to check a variable value, less often to observe an object's internal structure. Two programmers expressed confusion when they explored the internal structure of JavaScript objects: the JavaScript inspector exposes many internal object attributes (*e.g.,* __proto__ variables) that are usually not relevant to developers.

*Code snippet usage.* We observe two sessions where programmers wrote and executed code in a JavaScript internet browser console. In the first case, the developer was testing return values while sending methods to an object. In the second case, the developer wondered if sending a particular method to an object was properly handled. To obtain the object, he logged the variable to the console in his source code and refreshed the application. When the object appeared in the console, he used the "Store as Global Variable" feature to make the object accessible across runs. He then resumed his work, sending the method to the object several times.

*Debugger usage.* The debugger was used in only one session, where a participant put breakpoints in a web browser and observed variable values of his recently written program.

*Modifying running application.* One programmer modified the runtime state of his JavaScript program. He evaluated some code in a browser console and thus tested its correct behavior.

*Discussion.* We conclude that developers use feedback in their runtime environment; this includes executing applications and occasionally runtime state exploration. We think tool support has an important impact. We believe that participants used the HTML/DOM inspector iteratively because it is a mature solution: they can easily jump from HTML to source code, manipulate it and thus understand how it works. Similarly, the inspector integrated in web browser consoles was used frequently. But developers did not explore internals of JavaScript objects: understanding it is complex and tool support is insufficient. Further analysis is needed to confirm these implications and is out of scope of this paper.

*Summary: Developers in other languages use feedback when they can have it; more limited tool support restricts their options.*

## 8 FINDINGS AND DISCUSSION

We discuss our findings taking as our main point of comparison the study of Sillito *et al.* [39] and in particular the extended account found in Sillito's thesis [40]. This is the most extensive and in-depth account of a programming observation study that we know of. It also has the richest source of implications to which we can contrast findings. We also explicitly refer to other work as necessary.

### 8.1 On Liveness

*No silver bullet.* We agree with Tanimoto [8] that "*Liveness by itself is not a panacea for programming environment. Neither it is necessary for all programming tasks*". We find some instances where subjects were unable or unwilling to use liveness; they were subject to the same static exploration issues observed in traditional IDEs. Examples include P1 losing time in an overly exhaustive static exploration, and P2 missing valuable information from a too shallow one (similar to the "Inattention Blindness" phenomenon described by Robillard [38]). Missing knowledge, as observed by Murphy-Hill *et al.* [51] was a factor too, as we observed several instances where the halo or the debugger could have been used, but were not.

More generally, exploring unfamiliar code also imposed restrictions on how participants could obtain feedback, particularly with test cases (P4, P6), echoing observations by Sillito—writing test cases was the exception and in some cases challenging—and LaToza *et al.*, who in their study found that tests were not extensive enough and that developers could not rely on them [41].

In other cases, participants employed a variety of liveness techniques, but were not successful. P9 expected feedback, and lost time when he did not receive it. P4 kept several prototypes of solutions to his task as code snippets, experimenting with them iteratively, but did not succeed in his task as he "*did not want to play with threads, because it is too complex*", despite threads being necessary.

*Liveness eases checking assumptions.* Checking assumptions is important to ensure task progress, as Sillito observed: "*Programmers' false assumptions about the system were at times left unchecked and made progress on the task difficult.*" This was more likely when questions could not be answered directly by tools. In particular, Sillito found that setting breakpoints in the debugger was often used to assess if code was relevant for the execution of a feature. LaToza *et al.* [37] observed participants "gambled" when they did not have sufficient information, and noticed that "false facts" they inferred often led to further chains of "false facts".

Our participants were able to check their assumptions in a variety of additional ways, such as P11 quickly discarding an application layer from consideration thanks to a code snippet; or P6 finding out that two widgets were from different classes by inspecting them, and then finding functioning examples of the widget to infer the error came from its configuration, rather than the widget itself.

The participants were often able to quickly check their assumptions, ranging from the simplest (what is the class of this object?) to the more refined, thus spending less time on false tracks. When it was possible to progressively perform source code changes and test the behavior on running applications, participants did take full advantage of it. For example, participant P11 tested his work after *every small change*. Sometimes he expected a broken behavior, sometimes a positive progress; he checked regardless.

*Liveness eases refining context.* Context is a major hurdle that tools do not handle well, according to Sillito: "*A programmer's questions often have an explicit or implicit context or scope. [...] Tools generally provide little or no support for a programmer to specify such context [...], and so programmers ask questions more globally than they intend.*" The broadened scope makes information more likely to be missed and makes it even harder to check assumptions.

Pharo's static exploration tools have the same scoping issues as traditional tools. In fact, due to the lack of type annotations, scoping issues are worse than in statically typed languages [**? ? ?** ]. However, we have seen examples of liveness features being used to reduce the scope of investigations, such as P11 discarding an application layer thanks to a code snippet. A key point is that Pharo makes it easier to scope runtime questions as it is often *not necessary to run the entire application.* A developer can focus on an application's objects of interest (e.g., P9 inspecting a model and its view) and explicitly interact with those, for as long as necessary. Similarly, code snippets can call *parts* of existing code, and examples focus on a subset of an API.

*Some "hard questions" are easier to answer using liveness.* Sillito describes 44 questions developers asked during his sessions. While questions in categories 1 (finding focus points) and 2 (expanding focus points) all had full or partial tool support, questions in categories 3 (understanding a subgraph) and 4 (questions over groups of subgraphs) had mostly minimal, or at best partial support. These questions were higher level and the hardest to answer.

While liveness can not help for all questions, in some instances it definitely can. Question 27 (how does this data structure look at runtime?) is directly supported by the object inspector. Sillito found that 15% of sessions asked this question. All but one of our sessions (94%) used the inspector. Similarly, Question 37 (What is the mapping between these UI types and these model types?) can be easily answered by inspecting UI elements thanks to the halo.

Several questions in the fourth category concern comparisons between types, methods, or runtime behavior. Sillito states that "*Generally making comparisons is difficult*" and "*In some cases, even harder is a comparison between a system's behavior in two different cases as in question 34 (How does the system behavior vary over these types or cases?)*". One of Sillito's subjects wanted to "*figure out why one works and one does not*", and did two series of static reference navigations, comparing the differences. The participant obtained only a partial answer and missed the most important difference.

We found these kinds of comparisons in 12 out of 17 sessions, often thanks to multiple inspectors or snippets. Examples include P4 experimenting with multiple solutions in parallel, P6 inspecting two similar widgets for differences, P7 contrasting the behavior of two code snippets (one correct, one incorrect), or P11 contrasting an application with a snippet. While it is possible to open multiple debuggers at the same time, we did not observe this.

## 8.2 On Tool Support

*Tool support drives Live Programming.* Some major differences between this study and Sillito's stem from tool support. Sillito states "*[...] the questions asked and the process of answering those are influenced by the tools available [...]. Given a completely different set of tools or participants our data could be quite different.*"

We found evidence of this, as behavior that was easy to perform with Live Programming tools were frequently used. For instance, inspecting an object or executing an example are always one click away; so programmers used them very frequently. On the other hand, the Pharo search tools are fragmented: different tools perform different types of searches. The tools are both more difficult to use, and less frequently used. Knowledge is also a factor: accessing UI objects via halos was under-used, because many participants did not know the shortcut to activate it.

*Few searches.* Study participants used few searches, which contrast with studies that find code search to be a common behavior. Sadowski *et al.* [52] summarized these studies, finding code searching prevalent at Google. This finding agrees with Ko *et al.* [53], who found that developers preferred to use search tools that required less details. Another reason could be that users found other tools more effective, be it inspectors, debuggers, or browsing source code references. Ko and Myers observed in an experiment that users of the Whyline (with extensive runtime information) relied much less on search than users of their control group [54].

*Examples are more frequently used.* Sillito observed that in only 26% of sessions (7 out of 27), participants searched for examples (asking question Q4). The use of examples occurred in 59% of our sessions (10 out of 17). Pharo has a more prevalent "culture" of examples, including executable method comments, classes and methods marked as examples, and dedicated example browsers. Our survey offers additional insights. We find agreement with Sillito when he states that finding examples is challenging, as our participants indicated that finding relevant examples was more difficult than executing or modifying them. We think that the ease of execution of examples is a significant factor. It is harder to execute, for instance, a Java code snippet found on Stack Overflow, as it needs to be imported (resolving dependencies) and compiled before one can try it. Approaches facilitating this process, such as Ponzanelli *et al.*[55] or Subramanian *et al.* [56] should significantly ease these issues.

*Simple works best.* Some of the most advanced approaches were under-used by our participants. While many debuggers were opened (every runtime error in Pharo opens one), they tended to be quickly dismissed; comparatively few of them were used in depth, or explicitly opened via breakpoints. While several respondents mention that they value being able to "program in the debugger" in our survey's free-form comments, we observed none in our sessions. Similarly, we previously mentioned that writing test cases and using halos were under-used, the former as it needs extensive knowledge of the code to test, the latter for lack of awareness.

In contrast, the comparatively simpler tools such as the inspector and the playground were used by virtually all the participants. These tools are easily accessible: any piece of code can be evaluated and inspected. They are also very versatile, and this is partly due to the possibility to open multiple instances of the tools at once, and

keep them for a long period of time. Having several code snippets or inspectors open at once makes it easy to perform comparisons between different scenarios. Keeping these tools open for long periods of time makes it easier to check one's progress. Thus, combining these techniques yields powerful results, best exemplified by P7's use of two code snippets and two inspectors simultaneously.

*Comparison with online videos.* While we observed some liveness in the analyzed online coding sessions, we found much less usage than for Pharo participants. All of the online coding sessions used the facilities afforded by immediate feedback (either by manual reload of applications, or automatically after source code changes) to check progress on their tasks. Nearly all (10 of 11) sessions using JavaScript and HTML made use of the object inspector provided by the web browser. However, the JavaScript inspector usage was more limited since the tool support is less mature. Other usages of liveness were minimal. This confirms our previous observation that tool support is vital for people to make use of liveness effectively.

## 9 THREATS TO VALIDITY

Empirical studies have to make tradeoffs between internal and external validity [57]. This study's focus being on how developers used liveness in practice, we tried to maximize its external validity. To this aim, we performed our study in 3 steps.

We first observed 11 participants in 17 sessions who all used the same environment. To vary the situations, they worked on different tasks from ten different codebases. However, the duration of the tasks was short (40 minutes to an hour), which restricted the types of activities we observed. We could not do more for logistical reasons.

We further attempted to diversify our choice of settings by gathering 16 additional recorded coding sessions online, covering a variety of languages, settings, and tasks. We wanted to include sessions of the Jupyter interactive notebook [58], which has interesting features similar to the playground. We were ultimately unsuccessful: we found mostly tutorials, not real-world coding sessions.

We had limited choice of participants, and our participants were all male; studies like the ones of Beckwith *et al.* [59] have shown that males are more comfortable than females in tinkering. A Live Programming environment does encourage tinkering. In a follow-up study of debugging by end-users [60], females were found to be less frequent, but more efficient users of tinkering, showing that the relationship is not at all obvious. As such, studies of how females use Live Programming would greatly extend this work.

The tasks we investigated may not be representative or advantageous to a Live Programming environment, because such a representative task list does not exist yet. We only defined the tasks on unfamiliar code in the first study: we had no control on the tasks related to familiar code and the online video tasks, as both sets of tasks were defined by the participants themselves.

The transcription of the activity during the tasks was performed by the primary author. In such cases where the author was unsure, we held discussions on specific examples. To minimize biases in interpretation, most of the activity under study was related to tool usage (plus vocalizations), and we used Minelli and Lanza's DFlow to systematically record interaction data [48] (except in one session).

According to McDirmid, Smalltalk environments such as Pharo do not cover the entire spectrum of Live Programming (a specific example is provided by McDirmid [19]). We nevertheless chose Pharo as it is the "liveliest" environment used by practitioners that we had access to, and note that Pharo does include Morphic [29], which fits McDirmid's stricter definition of Live Programming. We also focus on liveness, rather than strictly Live Programming.

We kept the survey as short as possible to maximize responses. We had to make choices in what to ask and could not include every observation. Statistics omitted for lack of space—not respondents.

## 10 CONCLUSION AND IMPLICATIONS

The live feedback that Live Programming provides is gaining acceptance in mainstream tools. We performed a multi-phase study on how developers use *liveness* in practice. After observing 17 development sessions of Pharo developers, we concluded that *they used liveness very frequently*, and documented the various approaches they took. We found that participants favored *simple approaches*, and that liveness allowed them to *check their assumptions*. Advanced participants were able to easily compare and contrast execution scenarios; *practitioners may learn from the successful episodes we described.* A follow-up survey confirmed our observations on the frequency of usage of liveness. Finally, we contrasted our findings with online coding sessions in other languages, finding extensive use of immediate feedback, but limitations due to tool support.

*Implications for tool builders and language designers.* The major implication of this study is that *even small doses of liveness may have a large impact.* We saw that the most advanced approaches were sometimes not necessary, if simpler approaches such as inspecting could do the job. Technologically impressive tools such as the Whyline [54] have been developed, while the Debugger Canvas [61] or TouchDevelop [22] have been put in production. These tools are extremely useful and required a lot of effort to build. Yet, we think that adding a simple playground and inspector to an IDE could go a long way, particularly if multiple instances are allowed.

Similarly, Live Programming research focuses on languages with maximal liveness [6]. Yet we believe that exploring the design space of simpler representations such as the inspector and playgrounds could lead to significant improvements on its own, particularly if longevity and multiplicity are included. The notebook metaphor employed by Jupyter [58] elegantly allows for multiple code snippets (cells) to coexist, while providing easy visualization of the results (particularly for data), even if it is not fully live. The moldable inspector of Chis *et al.* [30] can be customized for each type of objects to display the most relevant information. Combining both approaches may be very intriguing.

*Implications for software engineering researchers.* The major implication is that there is still a lot to learn in how developers use liveness. As these features slowly but surely enter mainstream tools, multiple studies beyond this one are needed to fully understand how developers can best take advantage of the new capabilities their development environments will afford them.

Acknowledgment Juraj Kubelka is supported by a Ph.D. scholarship from CONICYT, Chile. CONICYT-PCHA/Doctorado Nacional/2013-63130188. We also thank Renato Cerro for his feedback and all the participants for their availability.

## REFERENCES


[1] K. Eliott, Swift Playgrounds—Interactive Awesomeness (2016).
URL http://bit.ly/Swift-Playgrounds

[2] C. Connect, Code Connect is joining Microsoft (2016).
URL http://comealive.io/Code-Connect-Joins-Microsoft/

[3] R. D. Team, React: A Javascript Library For Building User Interfaces (2017).
URL https://facebook.github.io/react/

[4] G. C. D. Team, Google Chrome Development Tools (2017) [cited August 2017].
URL https://developers.google.com/web/tools/chrome-devtools/

[5] B. Burg, A. Kuhn, C. Parnin (Eds.), Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013, IEEE Computer Society, 2013.
URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6599030

[6] S. McDirmid, The Promise of Live Programming, in: Proceedings of the 2nd International Workshop on Live Programming, LIVE '16, Rome, Italy, 2016.

[7] C. Parnin, A History of Live Programming (01 2013) [cited November 2016].
URL http://liveprogramming.github.io/liveblog/2013/01/a-history-of-live-programming/

[8] S. L. Tanimoto, A Perspective on the Evolution of Live Programming, in: Proceedings of the 1st International Workshop on Live Programming, LIVE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 31–34.
URL http://dl.acm.org/citation.cfm?id=2662726.2662735

[9] P. Rein, S. Lehmann, T. Mattis, R. Hirschfeld, How Live are Live Programming Systems?, Proceedings of the Programming Experience 2016 (PX/16) Workshop on - PX/16doi:10.1145/2984380.2984381.
URL http://dx.doi.org/10.1145/2984380.2984381

[10] E. Sandewall, Programming in an Interactive Environment: The "Lisp" Experience, ACM Comput. Surv. 10 (1) (1978) 35–71. doi:10.1145/356715.356719.
URL http://doi.acm.org/10.1145/356715.356719

[11] D. Ungar, R. Smith, SELF: the power of simplicity (object-oriented language), Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conferencedoi:10.1109/cmpcon.1988.4851.
URL http://dx.doi.org/10.1109/CMPCON.1988.4851

[12] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future, Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '97doi:10.1145/263698.263754.
URL http://dx.doi.org/10.1145/263698.263754

[13] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, T. Mikkonen, The Lively Kernel A Self-supporting System on a Web Page, Lecture Notes in Computer Science (2008) 31–50doi:10.1007/978-3-540-89275-5_2.
URL http://dx.doi.org/10.1007/978-3-540-89275-5_2

[14] J. Noble, S. McDirmid (Eds.), Proceedings of the 2nd International Workshop on Live Programming, LIVE 2016, Rome, Italy, 2016.

[15] D. Ogborn, G. Wakefield, C. Baade, K. Sicchio, T. Goncalves (Eds.), Proceedings of the Second International Conference on Live Coding, 2016.

[16] S. L. Tanimoto, VIVA: A visual language for image processing, Journal of Visual Languages & Computing 1 (2) (1990) 127–139.

[17] M. M. Burnett, J. W. Atwood, Z. T. Welch, Implementing level 4 liveness in declarative visual programming languages, in: Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on, IEEE, 1998, pp. 126–133.

[18] E. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, C. R. Cook, 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, 1997, pp. 258–265.

[19] S. McDirmid, Living it up with a live programming language, in: ACM SIGPLAN Notices, Vol. 42, ACM, 2007, pp. 623–638.

[20] J. Edwards, Subtext: uncovering the simplicity of programming, ACM SIGPLAN Notices 40 (10) (2005) 505–518.

[21] S. McDirmid, J. Edwards, Programming with managed time, in: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, ACM, 2014, pp. 1–10.

[22] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, J. Kato, It's alive! continuous feedback in UI programming, in: ACM SIGPLAN Notices, Vol. 48, ACM, 2013, pp. 95–104.

[23] R. DeLine, D. Fisher, B. Chandramouli, J. Goldstein, M. Barnett, J. F. Terwilliger, J. Wernsing, Tempe: Live scripting for live data., in: VL/HCC, 2015, pp. 137–141.

[24] Oracle, Java Platform Debugger Architecture (2014).
URL http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/

[25] S. Hanselman, Interactive Coding with C# and F# REPLS (2016).
URL http://bit.ly/InteractiveCodingCF

[26] K. Uhlenhuth, Introducing the Microsoft Visual Studio C# REPL (Nov 2015).
URL https://channel9.msdn.com/Events/Visual-Studio/Connect-event-2015/103

[27] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, 2009.
URL http://pharobyexample.org

[28] A. Goldberg, D. Robson, Smalltalk 80: the Language and its Implementation, Addison Wesley, Reading, Mass., 1983.
URL http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf

[29] J. H. Maloney, R. B. Smith, Directness and liveness in the morphic user interface construction environment, in: Proceedings of the 8th annual ACM symposium on User interface and software technology, ACM, 1995, pp. 21–28.

[30] A. Chiş, O. Nierstrasz, A. Syrel, T. Gîrba, The Moldable Inspector, in: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), Onward! 2015, ACM, New York, NY, USA, 2015, pp. 44–60. doi:10.1145/2814228.2814234.
URL http://doi.acm.org/10.1145/2814228.2814234

[31] A. Chiş, T. Gîrba, O. Nierstrasz, The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers, Springer International Publishing, Cham, 2014, pp. 102–121. doi:10.1007/978-3-319-11245-9_6.
URL https://doi.org/10.1007/978-3-319-11245-9_6

[32] J. H. Maloney, R. B. Smith, Directness and Liveness in the Morphic User Interface Construction Environment, in: Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology, UIST '95, ACM, New York, NY, USA, 1995, pp. 21–28. doi:10.1145/215585.215636.
URL http://doi.acm.org/10.1145/215585.215636

[33] A. J. Ko, B. A. Myers, M. J. Coblenz, H. H. Aung, An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks, IEEE Transactions on software engineering 32 (12).

[34] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, S. D. Fleming, How programmers debug, revisited: An information foraging theory perspective, IEEE Transactions on Software Engineering 39 (2) (2013) 197–215.

[35] T. D. LaToza, G. Venolia, R. DeLine, Maintaining Mental Models: A Study of Developer Work Habits, in: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, ACM, New York, NY, USA, 2006, pp. 492–501. doi:10.1145/1134285.1134355.
URL http://doi.acm.org/10.1145/1134285.1134355

[36] T. Roehm, R. Tiarks, R. Koschke, W. Maalej, How Do Professional Developers Comprehend Software?, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 255–265.
URL http://dl.acm.org/citation.cfm?id=2337223.2337254

[37] T. D. LaToza, D. Garlan, J. D. Herbsleb, B. A. Myers, Program Comprehension As Fact Finding, in: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, ACM, New York, NY, USA, 2007, pp. 361–370. doi:10.1145/1287624.1287675.
URL http://doi.acm.org/10.1145/1287624.1287675

[38] M. P. Robillard, W. Coelho, G. C. Murphy, How effective developers investigate source code: An exploratory study, IEEE Transactions on software engineering 30 (12) (2004) 889–903.

[39] J. Sillito, G. Murphy, K. De Volder, Asking and Answering Questions during a Programming Change Task, Software Engineering, IEEE Transactions on 34 (4) (2008) 434–451.
URL http://dx.doi.org/10.1109/TSE.2008.26

[40] J. Sillito, Asking and answering questions during a programming change task, Ph.D. thesis, University of British Columbia (Feb 2006). doi:http://dx.doi.org/10.14288/1.0052042.
URL https://open.library.ubc.ca/cIRcle/collections/831/items/1.0052042

[41] T. D. LaToza, B. A. Myers, Developers Ask Reachability Questions, in: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, ACM, New York, NY, USA, 2010, pp. 185–194.
URL http://doi.acm.org/10.1145/1806799.1806829

[42] A. Ko, R. DeLine, G. Venolia, Information Needs in Collocated Software Development Teams, in: Software Engineering, 2007. ICSE 2007. 29th International Conference on, 2007, pp. 344–353.
URL http://dx.doi.org/10.1109/ICSE.2007.45

[43] T. Fritz, G. C. Murphy, Using Information Fragments to Answer the Questions Developers Ask, in: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, ACM, New York, NY, USA, 2010, pp. 175–184. doi:10.1145/1806799.1806828.
URL http://doi.acm.org/10.1145/1806799.1806828

[44] E. Duala-Ekoko, M. Robillard, Asking and answering questions about unfamiliar APIs: An exploratory study, in: Software Engineering (ICSE), 2012 34th International Conference on, 2012, pp. 266–276.
URL http://dx.doi.org/10.1109/ICSE.2012.6227187

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

[46] J. Kubelka, R. Robbes, A. Bergel, Dataset for The Road to Live Programming: Insights From the Practice (Jan. 2018). doi:10.5281/zenodo.1156932.
URL https://doi.org/10.5281/zenodo.1156932

[47] W. Maalej, T. Fritz, R. Robbes, Collecting and processing interaction data for recommendation systems, in: Recommendation Systems in Software Engineering, Springer, 2014, pp. 173–197.

[48] R. Minelli, M. Lanza, Visualizing the workflow of developers, in: Software Visualization (VISSOFT), 2013 First IEEE Working Conference on, 2013, pp. 1–4. URL http://dx.doi.org/10.1109/VISSOFT.2013.6650531
[49] U. Flick, An introduction to qualitative research, Sage, 2014.
[50] F. Zieris, L. Prechelt, Observations on knowledge transfer of professional software developers during pair programming, in: Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on, IEEE, 2016, pp. 242–250.
[51] E. Murphy-Hill, C. Parnin, A. P. Black, How We Refactor, and How We Know It, IEEE Transactions on Software Engineering 38 (1) (2012) 5–18. doi:10.1109/TSE.2011.41.
[52] C. Sadowski, K. T. Stolee, S. Elbaum, How developers search for code: a case study, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 191–201.
[53] A. J. Ko, H. H. Aung, B. A. Myers, M. J. Coblenz, An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks, Software Engineering, IEEE Transactions on 32 (12) (2006) 971–987.
[54] A. J. Ko, B. A. Myers, Finding causes of program output with the Java Whyline, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 2009, pp. 1569–1578.
[55] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, M. Lanza, Prompter, Empirical Software Engineering 21 (5) (2016) 2190–2231.
[56] S. Subramanian, L. Inozemtseva, R. Holmes, Live API Documentation, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 643–652. doi:10.1145/2568225.2568313. URL http://doi.acm.org/10.1145/2568225.2568313
[57] J. Siegmund, N. Siegmund, S. Apel, Views on internal and external validity in empirical software engineering, in: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, Vol. 1, IEEE, 2015, pp. 9–19.
[58] M. Ragan-Kelley, F. Perez, B. Granger, T. Kluyver, P. Ivanov, J. Frederic, M. Bussonnier, The Jupyter/IPython architecture: a unified view of computational research, from interactive exploration to communication and publication., in: AGU Fall Meeting Abstracts, 2014.
[59] L. Beckwith, M. Burnett, S. Wiedenbeck, C. Cook, S. Sorte, M. Hastings, Effectiveness of end-user debugging software features: Are there gender issues?, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 2005, pp. 869–878.
[60] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, C. Cook, Tinkering and gender in end-user programmers' debugging, in: Proceedings of the SIGCHI conference on Human Factors in computing systems, ACM, 2006, pp. 231–240.
[61] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, S. P. Reiss, Debugger Canvas: Industrial experience with the code bubbles paradigm, in: M. Glinz, G. C. Murphy, M. Pezzè (Eds.), ICSE, IEEE, 2012, pp. 1064–1073.