Literate Exploratory Programming for Asynchronous Collaboration

Christoph Thiede
christoph.thiede@hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Tom Beckmann
tom.beckmann@hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Marcel Taeumel
marcel.taeumel@hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Robert Hirschfeld
robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
University of Potsdam, Germany

Abstract

A programmer can effectively explore systems by pursuing different approaches, conducting many experiments, and interacting with artifacts of the (runtime) environment. However, multiple programmers working together on the same exploration goal face challenges when communicating asynchronously, as they have to reproduce and explain relevant experiments and artifacts from their exploration to share them with coworkers. We propose *literate exploratory programming* as a novel workflow that combines exploration and explanation activities and reduces communication overheads through a tracking mechanism for extracting artifacts from the exploratory session. Inspired by traditional literate programming, in which implementation and contextualization are tightly interwoven, our approach introduces *exploratory journals*: a programmer can share artifacts such as code, runtime objects, and active debugging sessions together with prose, and another programmer can later resume the exploration from there. We demonstrate the feasibility of our approach by implementing a prototype in Squeak/Smalltalk and applying it for a design discussion in an open-source community. Our work indicates that tool support can foster effective collaboration in asynchronous exploratory programming contexts.

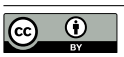
CCS Concepts

• **Software and its engineering** → **Collaboration in software development**; **Integrated and visual development environments**; Software prototyping.

Keywords

exploratory programming, literate programming, asynchronous collaboration, computational notebooks, debugging, Smalltalk

ACM Reference Format:
Christoph Thiede, Tom Beckmann, Marcel Taeumel, and Robert Hirschfeld. 2026. Literate Exploratory Programming for Asynchronous Collaboration. In *Companion Proceedings of the 10th International Conference on the Art, Science, and Engineering of Programming (‹Programming› Companion '26), March 16–20, 2026, Munich, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3801119.3801129



‹Programming› Companion '26, Munich, Germany

ACM ISBN 979-8-4007-2555-5/2026/03, https://doi.org/10.1145/3801119.3801129

1 Introduction

People often write code together. Students create prototypes in group projects, professional software developers build industrial solutions in teams, hobbyists maintain and extend open-source systems in global communities. Throughout this space, we observe various types of group dynamics: different team members bring in different fields of expertise, operate at different levels of proficiency, or focus on different goals. Likewise, collaboration styles and cultures vary widely: programmers can take individual code ownership for particular modules, look at the same screen during pair programming [2], or collaborate asynchronously across different time zones via mailing lists, communication platforms like GitLab, and so forth.

Imagine a typical situation in an open-source community: a user discovers a surprising error in an application and reports it in the project's bug tracker. Multiple developers start examining it by reproducing, debugging, and understanding the scenario. As they suspect the bug is not trivial, they join forces by gathering together their initial findings in the comments of the bug report. Eventually, they become able to explain why the error occurred, but the solution is still unclear: how should this special use case be covered in the architecture of the application? They dive deeper into the system and inspect related code, documentation, unit tests, or version control logs to learn about (implicit) assumptions, historic design decisions, and possible unintended consequences of new changes. They implement and compare different approaches—discussing the correctness and elegance of each—before settling on a final solution. During all of this, they perform numerous searches, inspections, and trials in the system and discuss their results in the bug tracker. Larger explorations might span several days or weeks [17], but the effort often pays off: solutions are grounded in well-founded evidence and reviews, and the collaborative process strengthens shared understanding and affiliative motivation within the community.

The *exploratory programming* practice can make debugging activities effective and enjoyable for individuals [23, 34, 38]: programmers repeatedly ask many questions, conduct many experiments, and interact with many artifacts from the system while gradually developing an understanding or a solution for a problem [45]. Programming environments can support them through helpful interfaces, tools, and metaphors: for example, Squeak/Smalltalk [19, 48] or Lively [20, 27] provide continuous and direct access to runtime

objects, along which programmers can navigate and change systems. In Jupyter notebooks [40], programmers run code and inspect results iteratively to analyze data or implement pipelines. In IDEs such as IntelliJ and Visual Studio Code, they navigate through the references, implementations, or occurrences of method names and read source code in tabs or panes.

However, when collaborating asynchronously, exploratory practices are impeded because knowledge handover becomes challenging. Programmers often wish to cooperate closely and to have frequent, rich exchanges, in which they discuss steps and ideas even before they lead to mature solutions [28]. They need to transfer detailed knowledge of experiments, observations, and thoughts to the machines and minds of their peers. This is mentally and technically challenging: thoughts can be scattered, chaotic, and difficult to explicate until they form coherent and sound ideas. Similar to this intangible knowledge, tangible artifacts such as code snippets, outputs, or debugging states from the ongoing exploratory session of a programmer are often scattered on their machine. To explain their meaning or context and enable collaborators to reproduce them, programmers need to isolate, clean up, and order artifacts. All this communication requires time and mental effort, making close collaborative exploration costly and thus preventing better or faster solutions [54].

Literate programming addresses the problem of insufficient or decoupled communication in traditional software development by tightly interweaving implementation and explanation [25, 31, 32]. In computational notebooks, a modern instance of literate programming environments, programmers alternate inputs, outputs, and prose to explain a data analysis or prototype to human readers [24]. Yet, a focus on communication and explicating thoughts might collide with productive exploratory programming: exploratory programmers want to branch out and freely explore different options instead of structuring and explaining every step while the merit of the step is still unclear. They want to iterate quickly, follow their intuition, maintain their *flow* [11], not to note down reproducible experiments or provide cumbersome context continuously. Still, we argue that unbound, cheap exploration and effective collaboration do not contradict each other but require better integration in the overarching programming process. This leads to our research question:

> How can we improve asynchronous collaboration for exploratory programmers without disrupting their exploratory flow?

We propose *literate exploratory programming* as a novel workflow that combines exploration and communication. To this end, we extend traditional exploratory programming environments with a means for tracking and documenting experiments and artifacts in an *exploratory journal*. We believe that these new capabilities of programming environments allow programmers to focus on their exploration and reduce the overhead for switching between exploratory and communicative activities.

In the following, we analyze the tension between exploration and explanation that programmers face (section 2), describe our approach for a literate exploratory programming workflow (section 3), present a prototypical implementation for Squeak/Smalltalk (section 4), and discuss the potential of our approach through an experience report (section 5) and compare it to related work (section 6) before we conclude (section 7).

2 Building a Bridge Between Exploration and Explanation

A good exploratory programming session is like an intensive conversation between a programmer and a software system [45]. This conversation is mediated through the tools and interfaces of the programming environment. Similar to science, this conversation may appear unstructured and structured at the same time: at large, the programmer follows their curiosity, while at small, they apply the scientific method—continuously posing new questions or hypotheses; looking up information or conducting experiments in the system to gain answers; interpreting the results; and repeating [49]. Knowledge emerges gradually during this conversation. An (improved) theory of some behavior is formed, a prototype grows, a system is extended or refactored.

Conversations should be fruitful and vivid. Programming environments can contribute to that experience by acting as almost transparent mediators between the programmer and the system that provide direct, *immediate* [51] access to its artifacts [34]. For example, in Squeak/Smalltalk, code execution is ubiquitous [19, 48]: when we see a method in a *system browser*, we can hit `[CMD]+[D]` to execute it in place ("do it"). We can open the returned result object in a new *inspector* to examine its variables or properties. From the inspector, we can also execute code snippets in the context of the object (i.e., *sending messages to objects*). Through another shortcut, we can even bring up an ad-hoc *debugger* to explore the execution of that snippet.[1]

Suitable environments for exploratory programming support *multitasking*, since exploration is rarely linear: while conducting an experiment, we might ask further questions about subproblems [47]. Maybe we want to compare different artifacts with each other. Maybe a fresh idea distracts us, and we try out that approach first, following the path of least resistance to the desired answer [33]. In Squeak, artifacts are presented in small, fine-grained, non-modal windows that programmers can freely switch between [44]. Programmers often keep dozens or even hundreds of windows in their environment during an exploratory session [36, sec. 5.2; 29]. Like notes on a desktop or post-its on a whiteboard, these windows can also be juxtaposed on the screen. This flexible arrangement allows programmers to align them with their mental models, make sense of artifacts by viewing them side by side, or quickly organize their environment for explaining something to an office mate [8].

2.1 Communication Challenges

When our exploration comes to an end, or we find ourselves stuck or simply wish to call it a day, we want to hand over our current progress to a (remote) coworker. This coworker should be enabled to understand, potentially reinterpret, and resume the exploration

[1] In Smalltalk, execution is modeled as first-class process objects akin to green threads, allowing multiple debugging sessions to exist in parallel without constraining other artifacts in the environment.

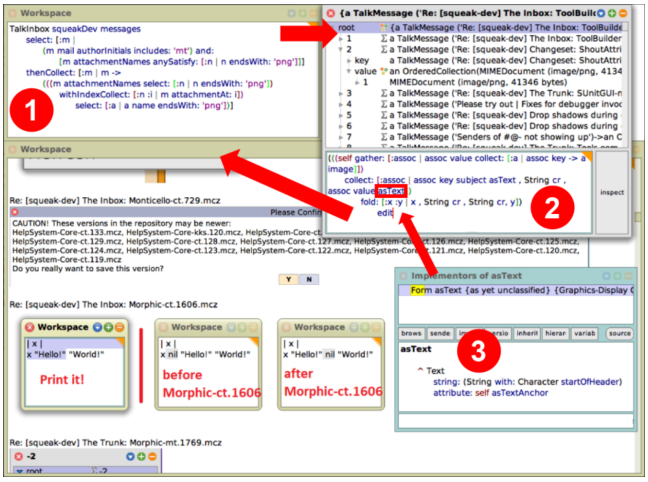


Figure 1: Screenshot of artifacts arranged in Squeak with the goal of explaining a simple exploration. Red overlays added by the programmer (arrows, numbers, text) hint at the causal order of involved artifacts and experiments. Here, the programmer has compiled an overview of images from the squeak-dev mailing list by combining short scripts, exploring runtime objects, and defining a convenience method for integrating images into text.

session as if they had sat next to us for the last few hours. To this end, we need to share with them:

- Our ideas and approaches
- Experiments we ran, including instructions on how to reproduce them
- Relevant artifacts we discovered or created
- (Preliminary) conclusions and next steps

Compared to face-to-face conversations, asynchronous communication greatly reduces the bandwidth and interactivity of communication [13, 14]. So, all that information needs to be provided in a form that is *distilled*, *self-explanatory*, and offers a clear *reading order* [15] to guide the coworker through the different paths of our exploration.

How can we convey all this knowledge to a coworker? A first step could be to write down a report of our exploratory session and thought process as continuous prose. To enable the coworker to reproduce the artifacts we have used, the prose should include code snippets and pointers to classes or methods of interest. However, producing and consuming long, abstract texts multiple times a day is ineffective for both involved parties: we need to manually detail how each scenario was set up, each object was created, each debugging session was started; they need to closely follow our instructions to restore any artifacts they want to explore further.

Alternatively, we could also share screenshots or even screencasts of tools and artifacts from our workflow. While recording a quick screenshot is done with a single shortcut, the result may leave the receiver puzzled with a crowded photo lacking a clear reading order. To make this usable, we would have to spend additional time splitting up and rearranging large screenshots or drawing additional explanatory annotations onto them (fig. 1). Although it is possible to automate such annotations and arrangements [3], the receiver still cannot easily and reliably reproduce the displayed artifacts themselves. Screencasts, on the other hand, add new challenges such as performing and comprehending a sequence of actions in real time and cutting or replaying the video.

Some environments also support creating and sharing snapshots of the entire programming session. For instance, Smalltalk environments can be saved as images or project files [43], which contain the entire object graph, including open tools and debugging sessions to be resumed later; other IDEs such as Eclipse or Emacs store a list of open files and their UI state in workspace files. However, all of those snapshots tend to be chaotic and contain much noise, such as irrelevant artifacts and attempts or user-specific settings. Like with screenshots, we still would need to provide additional structure and explaining context in this approach. So, we are looking for a semi-automated approach that helps us track exploratory workflows and also curate (explain) extra details so that fellow programmers can comprehend our insights and progress from our exploration.

2.2 Literate Programming to the Rescue?

To better connect such artifacts to technical context and to our ideas, *literate programming* proposes that we develop both at the same time: we can create an executable document and fill it with an alternating sequence of technical and non-technical elements [31, 32]. In its earliest forms, literate programs were written to optimize the readability of programs for humans, not computers [25] and establish a reading order for non-linear source code [1]. To also explain runtime observations, we can integrate outputs and intermediate results of programs into *computational notebooks* [24, 40] or *active essays* [26, 56]. Readers and writers of a notebook can interact with these artifacts to test or incrementally evolve the program. Similarly, we can also write literate programs that employ scriptable debugging [30] to execute another program step-by-step, record runtime values or executed methods like in a program trace [50], and explain our observations through interspersed prose [41, 42].

Still, the practice of literate programming substantially restricts the exploratory programming experience: first, programmers are often overwhelmed by the amount of automatically collected experiments and artifacts when navigating between different artifacts to compare or interact with them: for example, users of computational notebooks report overheads for scrolling through or cleaning up irrelevant cells [24, 37]. Second, the linear style of literate programming prevents programmers from multitasking effectively. Notebook users frequently lack options for exploring alternative branches [53], resort to stacking chronologically unrelated experiments on top of each other [24, 37], and subsequently lose the ability to rerun their notebooks to reproduce experiments [37, 53].

So, programmers face a dilemma between optimizing their workflow for exploration or communication: if they rely on rich exploratory environments that support unbound navigation, they need to manually document, snapshot, and organize artifacts and experiments. If they adopt a literate programming style instead, they will lose multitasking and spatial flexibility during the exploration.

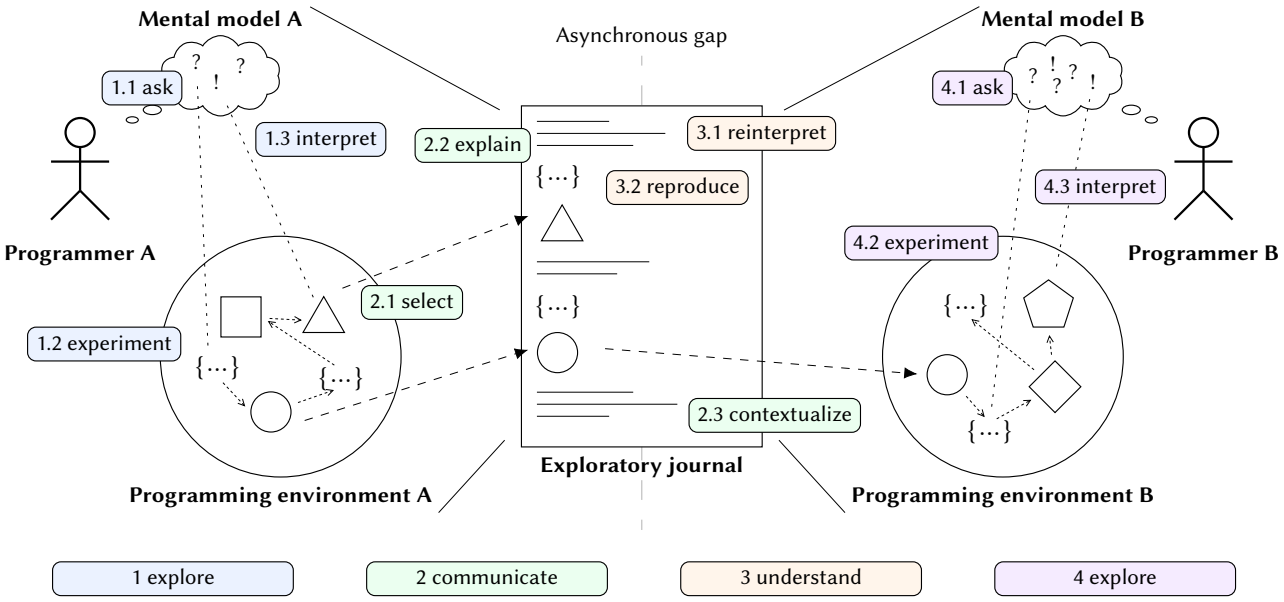


Figure 2: The literate exploratory programming workflow. Two programmers collaborate across an asynchronous gap by cycling between exploration, communication, and understanding. During exploration, they interact with the artifacts in their local programming environment to develop ideas and solutions. To communicate their exploratory session, they select relevant artifacts, arrange them in the exploratory journal, and explain them through prose. To resume the exploratory session, another programmer can read the journal, reproduce experiments, and interact with the saved artifacts in their programming environment. The programming environment supports communication by tracking all experiments in the background and inserting them into the journal to ensure reproducible artifacts.

3 Literate Exploratory Programming

We propose *literate exploratory programming* as a new workflow that augments exploratory programming with means for capturing and explaining artifacts. In this workflow, two programmers collaborate asynchronously by alternating between an *exploration* and a *communication* phase (fig. 2): during exploration, they interact with their exploratory programming environment to gather artifacts and insights. For communication, they exchange the current state of their exploratory session through a shared *exploratory journal*. The programming environment continuously monitors the activities of the programmer and allows them to insert and document relevant slices of it into the journal. Thus, literate exploratory programming preserves unrestrained multitasking and spatial arrangement from traditional exploratory programming, while allowing programmers to efficiently communicate the exploratory session in a linear form.

Our workflow describes the interactions between three sites:

Programmer A: owns a *mental model* with ideas and an *exploratory programming environment* to conduct experiments and interact with artifacts.

Exploratory journal: consists of prose, experiments, and artifacts. Will be exchanged between both programmers.

Programmer B: owns a separate *mental model* and *exploratory programming environment*.

Both programmers go through four stages repeatedly:

(i) Local exploration Programmer A begins an exploration session in their programming environment. They create and collect artifacts, interact with them through tools, interpret them to form ideas, and switch or navigate between different artifacts as desired. In the background, the programming environment automatically constructs a dependency graph to track the provenance of any encountered artifacts and experiments.

(ii) Communication Programmer A selects tools from their environment that display relevant artifacts to communicate their current progress. The environment appends snapshots of the selected tools to the exploratory journal and prepends a chain of underlying experiments that the programmer has performed earlier to create these artifacts. The programmer then inserts context and ideas as prose around the artifacts in the journal. Optionally, they can also revise experiments or rearrange artifacts. Finally, they share the journal with programmer B.

(iii) Understanding Programmer B receives the journal and loads it. Their programming environment automatically reproduces all artifacts from the journal by rerunning experiments. The programmer reinterprets the prose, experiments, and artifacts by reading the journal and interacting with the snapshotted tools.

(iv) Continued exploration Programmer B spawns artifacts of interest from the journal into separate tools in their environment. Thus, they branch off from the reading order of the journal, review the thoughts of programmer A, and answer follow-up questions through new experiments. If they wish to communicate new progress back to programmer A, they can repeat from step (ii) with reversed roles.

Thus, an exploratory journal resembles a computational notebook that can contain arbitrary runtime tools such as inspectors and debuggers. Still, programmers in our workflow only use journals for communication, while their exploration takes place in the surrounding programming environment.

Figure 4 (appendix) shows an example of an exploratory journal from our prototypical implementation in Squeak/Smalltalk. In this example, we have investigated a bug in Squeak's process termination logic (based on the scenario described in section 1). Initially, we had debugged a failing test multiple times; tried out different variations of the test; inspected several instances of the invalidated process object at different points in time during the termination; and studied a lot of related methods and class comments in Squeak's Kernel package to gather an understanding of the nature of the bug. After we had discovered that the bug is a race condition between two parallel stack unwinding routines, we documented our findings in an exploratory journal: from the numerous open windows in our environment, we selected the original test method, a minimal script to trigger the bug, a number of meaningful stack frames from the process termination, and snapshots of relevant runtime objects; arranged them in the journal; and added detailed commentary to explain the different steps of our investigation, our results, and open questions.

Another collaborator then opened our journal and followed our findings. Some of our steps and deductions became clear to them immediately, while they were puzzled about others or questioned them. To review those parts, they could inspect some of the recorded runtime objects in external windows to explore their properties, resume one of the snapshotted debugging sessions in their environment, step further to another method, and eventually prove or falsify our hypotheses. Next, they could expand on our investigation by thinking about possible fixes of the race condition, implementing some solutions, and testing them by rerunning our journal. Once they have reached a satisfying proposal, they can add their patch, a demonstration of its design and functionality, and their considerations to the journal and send it back to us.

4 Implementation

In the following, we describe the technical details for integrating literate exploratory programming into a traditional exploratory programming environment. The programming environment needs to support two new capabilities:

(i) A tracking mechanism that records dependencies between artifacts and experiments and allows programmers to extract relevant artifacts.

(ii) An exploratory journal that embeds prose, executable experiments, and tools for recorded artifacts.

For our experiments, we implemented a prototypical extension for literate exploratory programming in Squeak/Smalltalk. We make the source code and all examples available online.[2]

Tracking mechanism. To capture the context of artifacts, we associate each tool with a linked list of navigation and experimentation steps. Each step can either be a *do-it* (a code snippet) executed by the programmer or a *message* (a method call) that was sent to the artifact from the environment. We patch standard tools of the environment such as workspace, inspector, explorer, and debugger to track these steps when the programmer modifies an artifact or opens a new tool.

To track code execution, we hook into the code editor through callbacks such as `#doIt:result:`, `#inspectIt:result:`, etc.

To track navigation through the object graph of the environment, we modify relevant triggers for new tool invocations in inspectors and debuggers. Simple variable accesses can be mapped to message sends of the form `self instVarNamed: 'variableName'` or, for array fields, `self at: 1`. For domain-specific (moldable [10]) accesses such as convenient property lists, aggregations, or example instantiations, we use different metaprogramming strategies: for inspectors, we can decompile the registered `valueGetter` block closure of a field and substitute captured variables with their runtime values. For explorers, we use a custom message interception technique to wrap return values of methods into transparent decorators (see section A). For navigation in debuggers, we record every step such as `#stepInto` or `#stepOver` as a message sent to the debugger object.

When our patches miss an interaction in an otherwise instrumented tool, the journal will not include the step but there is currently no mechanism besides a manual check for the user to learn that this is the case. While adding tracking support for further tools may require some effort to reverse-engineer their implementation or update patches for newer tool versions, we did not encounter any fundamental limitations to our tracking framework thanks to the open nature and metaprogramming facilities of Smalltalk.

When tracking is enabled, we add a new record button to the title bar of every `SystemWindow` (shown in 2, 7, 8 of fig. 3). When the programmer clicks it to record a tool, we insert all associated experiment steps and a copy of the tool into the journal.

Exploratory journal. We implemented exploratory journals as a variation of Squeak's default workspace tool (shown in the right half of fig. 3). A workspace is a simple text buffer that programmers can use like a scratchpad to execute code snippets, create variables, and display inline results ("print-its")[3]. In our journal, experiments are expressed as Smalltalk code to set up artifacts and configure tools through a tiny DSL. Prose is written as comments. Artifacts are displayed through customized print-its: we replace the default textual description of each result with a `TextAnchor` for inlining a compacted, interactive instance of the respective tool. Through buttons, the programmer can run or rerun all experiments up to an artifact; spawn an artifact as a separate window into their programming environment; and export the entire journal as a file to share it.

For export, we serialize the text of the journal as HTML with a custom `<ExploratoryJournalTool>` tag that describes the UI state of each tool. This tag falls back to a nested ordinary `<img src="...">` tag with a screenshot of the tool, making the exported file readable in web browsers as well. When opening a journal file, the `TextAnchor` for a tool displays the screenshot as a placeholder until the programmer revitalizes the artifact by rerunning the preceding experiments.

[2] Source code, online playground, and screenshots: https://github.com/hpi-swa-lab/exploratory-notebooks
Screencast: https://doi.org/10.5281/zenodo.18889419

[3] https://wiki.squeak.org/squeak/1934 (accessed at 2026-02-07T00:45Z)

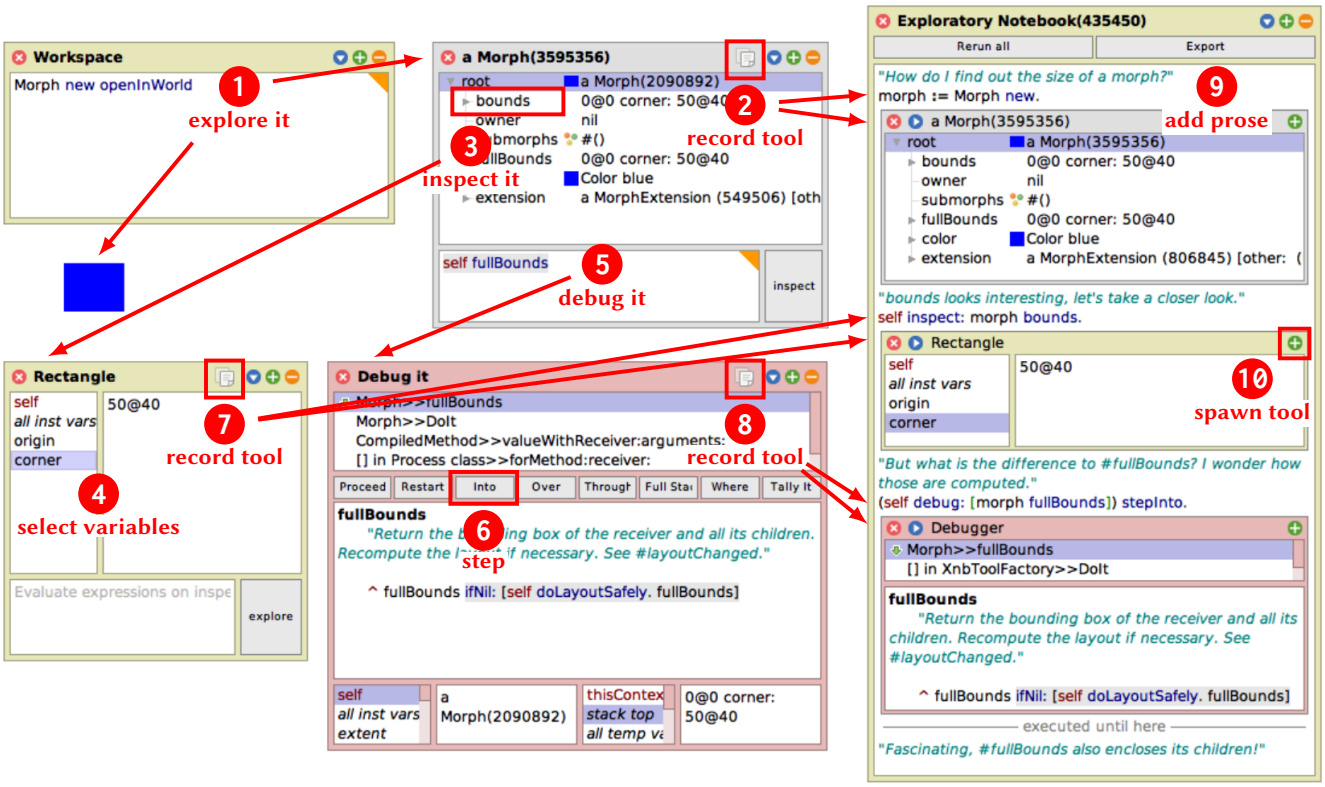


Figure 3: Example interaction with our prototype for literate exploratory programming in Squeak. In the four windows on the left, a programmer is performing an exploratory programming session through standard tools such as workspaces, inspectors, and debuggers. These tools have been augmented with (a) a transparent tracking mechanism to log all experiments and artifact dependencies and (b) a record button added to every tool window. On the right, the programmer curates an exploratory journal by collecting relevant experiments and artifacts through the record buttons and explaining them with additional prose. Another programmer can later reopen the journal, rerun it, and spawn tools in their environment to retrace and resume the exploration.

Interaction. Figure 3 shows a simple example of an exploratory session in Squeak during which a programmer records relevant artifacts into our prototypical journal. The programmer begins by writing and running scripts in an ordinary workspace to create a graphical object and interact with it (1). As soon as they start their exploration, the instrumented programming environment automatically tracks all their steps in the background. Our programmer opens the created `Morph` object in an instance of Squeak's explorer tool and examines its state by expanding the property tree. As they decide they want to record this tool, they press the "add to journal" icon in the title bar of the explorer window (2). This opens a new exploratory journal and adds a copy of the explorer tool to it, prepended by the tracked code that the programmer executed before to create the tool.

Next, the programmer continues with their exploratory session: for instance, they can further navigate through the object tree, invoke another inspector window from one of the `Morph`'s properties (3), and examine the variables of that property there (4). In parallel, they could also launch a debugger from the explorer (5) and step into a message send to explore how the `Morph` handles it (6).

As the programmer invokes new tools or navigates through different contents in them, the environment keeps tracking all their steps but does not automatically add any experiments or artifacts to the journal. Only if the programmer deems particular steps or observations relevant, they can press the record button in any tool window to insert a copy of that tool plus all preceding experiments that the current artifact in the tool depends on into the journal (7, 8). This allows them to focus on their exploratory session without constantly recording relevant artifacts, as all tool windows remain open until they clean up and possibly record a subset of them at the end of their session. By default, new tools and experiments are appended to the end of the journal, but the programmer may reorder them for their explanation while maintaining a topological ordering of dependent code snippets. Finally, they can add explaining context to the journal by decorating the recorded experiments and artifacts with additional prose before they export the journal to share it (9).

Later, another programmer can reopen the shared journal and read it. If they click on the spawn button of any embedded tool, the relevant snippets of the journal will automatically rerun and a copy of the tool will open as another window in the environment (10).

5 Discussion

Although we have applied literate exploratory programming only to a small number of practical use cases so far, we are already positive about our initial experiences: multiple programmers can effectively explore systems in an asynchronous collaboration style without having to manually replicate each other's experiments and artifacts. Using the tracking mechanism, they can communicate more easily and frequently by creating structured, reproducible explanations of their experiments *ex post*. Programmers retain the well-known features of an exploration-centric environment such as unbound navigation, spatial flexibility, and a rich, extensible toolset, even when they are unsure whether they will communicate their results to others later.

Exploratory journals facilitate comprehension of a shared session through a high degree of *spatial immediacy* [51] between related experiments, artifacts, and thoughts. When programmers switch between exploration and communication, journals also preserve *semantic immediacy* by displaying embedded artifacts using the familiar tools of the programming environment, enabling consistent and easy perception of domain objects [6].

By explicitly curating experiments and artifacts, programmers can summarize large tracked exploratory sessions into manageable and concise journals. For example, our initial exploration of the scenario described on page 5 spanned multiple days and involved hundreds of experiments and artifacts, which we could condense to 13 essential snapshots for the final journal in fig. 4 that were sufficient to enable a coworker to understand our findings.

5.1 Limitations

Despite all the above, our approach makes a few specific assumptions about the style of the exploratory process and the existing environment: explored setups must be reproducible. For instance, when programmers interact with an application and encounter a runtime error, they still need to reproduce it before they can share their insights from debugging it with others as a constructive bug report [5]. While we could address this limitation by recording all interactions and side effects in the runtime using an upfront program tracer [50], this would pose a new challenge: how can we identify relevant side effects for a selected artifact to avoid polluting the exploratory journal with many irrelevant code snippets?

We also assume a direct mapping between observations and artifacts that fit into a concise journal without overwhelming: Smalltalk systems present methods and runtime objects in fine-grained tools, but in other cases, large monolithic tools provide access to convoluted information such as profiler hierarchies and aggregated dependency graphs.

Our approach focuses on explanation as a linear process but does not make the recorded artifacts in a journal *directly manipulable* [39]: to change recorded artifacts, programmers have to spawn embedded tools into their environment, interact with them, and only then re-record them. Alternatively, they can manually edit the code for creating tools. The same is required if they wish to optimize the readability of experiments: while the tracking mechanism reliably allowed us to reproduce interactions in our initial tests, the generated code might look redundant or obscure. Future work should explore how user intentions can be reconstructed after tracking—for instance, to rewrite a long sequence of steps in a debugger into a single "step into" or "step until" command—, potentially applying methods from *programming by demonstration* [12].

Ultimately, communication is an integral challenge in col-*labor*ation that distinguishes it from plain (individual) labor: even with the recent rise of generative AI [46], the act of explicating your own mental model remains one of the last responsibilities that cannot be delegated entirely to tools.

5.2 Future Work

To evaluate the practical value and limitations of our approach, we next want to implement literate exploratory programming into our everyday collaborative processes and learn from our experiences of asynchronous exploration with peers. To this end, we are planning to integrate our prototype more deeply into Squeak by supporting advanced tools such as code reference search, profiling, and back-in-time debugging. By making available exploratory journals through Squeak's built-in mailing list interface[4], we hope to study how they can support collaboration patterns in an open-source context.

Beyond asynchronous collaboration, we envision several other applications of literate exploratory programming for supporting communication patterns in exploratory contexts. In the rest of the section, we highlight four interesting avenues of further research.

Self-organization. Even when programmers explore on their own without collaborating, asynchronous elements in the process remain: programmers switch between multiple tasks, go into the weekend, and later resume their own exploration after a mental context switch. In one case, we used an exploratory journal as a (semi-structured) "brain dump" to explain to our "future selves" where we had stopped. While creating that journal, we also experienced that this helped us structure our own thoughts and reflect on our process and findings.

Real-time collaboration. So far, we have only considered asynchronous collaboration settings. However, sharing artifacts and experiments across multiple devices in a synchronous exploration session is challenging, too. We think that our proposed tracking mechanism could aid the construction of a collaborative exploratory environment where programmers have direct access to the tools and experiments from their collaborators.

Documentation and training. Another idea is to use exploratory journals to provide programmers with interactive documentation or tutorials. For example, such journals could present code snippets, runtime objects, or debugging slices to explain how a system works, how a library can be used, or even how an architecture has evolved [55], while inviting readers to branch off from the narrative and explore their own questions.

Next to educating readers about particular domains, exploratory journals could also be used to teach novices about exploratory practices. From our experiences with teaching students to program in Smalltalk, we know that many students struggle with adopting exploratory practices in their own workflow. "Exploration by example" journals could become another element in the toolbox of

[4] https://github.com/hpi-swa-lab/squeak-inbox-talk

teachers between demos in lectures and project-based learning for encouraging students to use programming tools effectively and explore systems through runtime inspections.

Narrative structures. Expanding the role of exploratory journals for communication could also raise the need for richer forms of presentation: could we make journals more *interactive* by drawing inspiration from concepts such as active essays [56] or hermeneutica [35]? How could we express multiple *alternative* (perhaps failed) approaches from an exploration or variations of artifacts [4, 22] in a nonlinear, yet structured form? How far can a journal *scale* in terms of length (missing structure) and longevity (surviving refactorings)?

6 Related Work

Prior work has already proposed different strategies to support the reconstruction of program exploration and exploratory programming sessions. *Mylar/Mylyn* tracks which elements of a project a programmer opens or edits frequently while working on a task and later suggests these and related elements to the same programmer to help them navigate through large software systems [21]. *Code gathering tools* track the execution and dependencies of code snippets while programmers are working in a computational notebook. When a programmer wants to share their work, they can select a set of artifacts from the notebook and generate a distilled program slice of relevant code snippets [18].

While this technique addresses the problem of unclear execution order in computational notebooks, exploration remains limited to a linear arrangement of code and artifacts. Other works propose nodes-and-wires interfaces for exploratory programming to overcome this limitation and allow programmers to experiment with different variations and combinations of code snippets [7]. Still, to the best of our knowledge, no prior work has studied the idea of tracking activities and artifacts from program exploration, prototyping, and debugging in a holistic manner with the goal to enable communication and later resumption of an exploratory programming session.

7 Conclusion

We presented our concept of *literate exploratory programming* and its implementation for the Squeak/Smalltalk programming environment. Literate exploratory programming allows programmers to consolidate a branching exploratory session into a linear *exploratory journal* that intersperses a programmer's interactions in a programming environment with prose to explain the actions and their results. The resulting journal can be sent to another programmer who can read the rationale and resume the exploration activities embedded by the original author. In first tests, we demonstrated the concept's technical feasibility and its suitability for collaboratively investigating a bug in Squeak itself. We consider exploratory journals to be a promising bridge between computational notebooks and unbound exploratory environments, supporting both unrestrained exploration and its communication through explicit structuring of rationale.

Acknowledgments

We thank Jaromír Matas for engaging with us in countless collaborative exploratory sessions on the Squeak developers' mailing list, which has motivated the idea in this paper. This work was supported by the HPI Research School on System's Design[5], SAP, and the HPI–MIT program "Designing for Sustainability"[6].

A Appendix: Tracking Object Accesses Through Message Interception

Here, we describe a message interception technique for Smalltalk that we used in our prototype (section 4) for tracking message sends to artifact objects from domain-specific tools to display custom representations. To our knowledge, this specific approach has not been described before.

When programmers interact and navigate with artifacts through domain-specific tools, such as third-party extensions to the programming environments, these tools may access runtime objects in a way that is unknown to the implementor of the tracking mechanism. At the same time, we do not want to put the burden for tracking these accesses on each domain-specific tool. For example, the hierarchical contents in Squeak's object explorer can be modified by overriding the method `#explorerContents`:

```
XMLElement»explorerContents
    ^ self attributes keys collect: [:key |
        ObjectExplorerWrapper
            with: (self attributes at: key)
            name: (key asString contractTo: 32)
            model: self attributes]
```

Here, the tool extension hardcodes the access `self attributes at: key` to an XML object, and the object explorer—that we could modify for tracking—only sees the resulting value, such as another XML object. This distinguishes the object explorer from Squeak's object inspector, where tool extensions need to specify explicit callbacks or closures, which we could decompile.

Our solution to this problem combines elements from membrane proxies [52] and method wrappers [9, 16]: before we invoke the domain-specific tool, we wrap the object into a transparent proxy that tracks all accesses to the object. When a message is sent to the proxy, the proxy forwards it to the object and wraps the result in another proxy, which remembers the original message in a list.

However, relevant results are often composed from multiple messages or closures, such as the `ObjectExplorerWrapper` instances in the above example, which are created from the `#collect:` send and the `[:key | ...]` block. To track the origin of such parts, we do not send forwarded messages directly to the target object but manually look up the respective method and execute it against the proxy. Because Smalltalk inlines instance variable accesses and special message sends in the bytecode of methods, we exclude methods with these features from this mechanism.

The following shows the essential logic of our tracking proxy:

[5] https://hpi.de/en/research/research-schools/systems-design/
[6] https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html

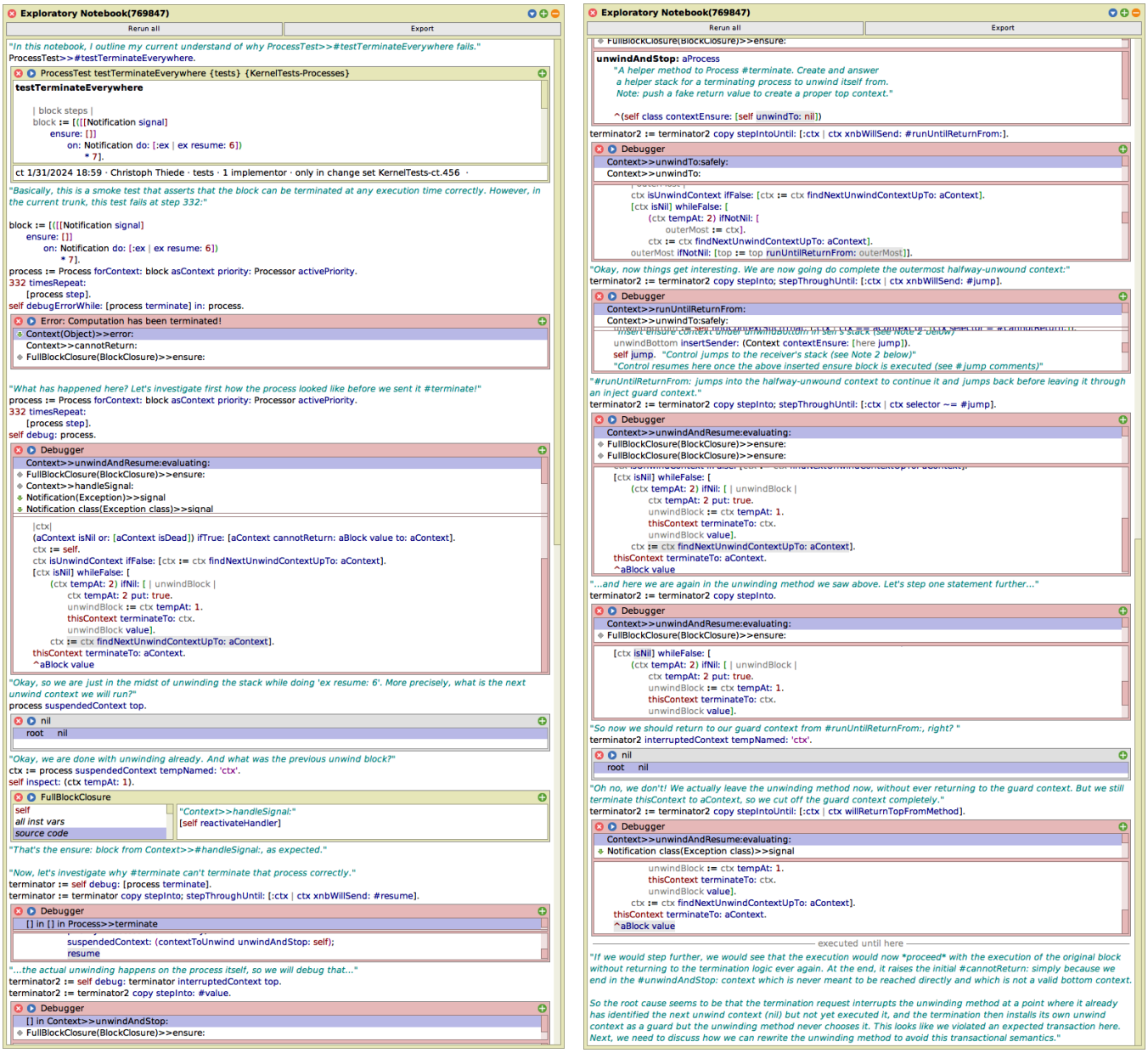


Figure 4: Example of an exploratory journal documenting a real debugging session from the example on page 5. In this scenario, we investigated a failing test in Squeak's process termination engine. Note that the tools embedded in this notebook text document are interactive and fully functional.

```
ProtoObject subclass: #TrackingProxy
  instanceVariableNames: 'target trackedMessages'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'TrackingProxy'.

TrackingProxy»doesNotUnderstand: aMessage
  | lookupClass selector arguments method result |
  lookupClass := thisContext objectClass: target.
  selector := aMessage selector.
  method := (lookupClass lookupSelector: selector) ifNil:
    [arguments := aMessage.
     lookupClass lookupSelector: #doesNotUnderstand:].

  method methodClass ifNotNil: [:methodClass |
    (methodClass includesBehavior: SmallInteger) ifTrue:
      ["Prevent special message sends for SmallInteger (e.g. #<=)"
       method := method copy
         literalAt: method numLiterals put:
LargePositiveInteger binding; "has the same superclass"
         yourself]].

  result := method
    valueWithReceiver:
      ((method primitive ~= 0 or: [method hasInstVarRef] or: [method
sendsToSuper] or: [method sendsSelector: #class])
        ifTrue: [target] ifFalse: [self])
    arguments: (arguments ifNil: [aMessage arguments]).

  ^ result xxxWithTrackedMessage: aMessage

Object»xxxWithTrackedMessage: aMessage
  ^ TrackingProxy for: self trackedMessages: aMessage

TrackingProxy»xxxWithTrackedMessage: aMessage
  ^ TrackingProxy for: target trackedMessages:
    (trackedMessages copyWith: aMessage)
```

We can now inject this proxy into a domain-specific tool as follows:

```
(TrackingProxy on: anXmlElement) explorerContents.
```

The resulting object graph can be displayed regularly in the UI as if it consisted of actual XML nodes and strings, and the programmer can continue to navigate and experiment with the returned XML structure wrapped in proxies. However, whenever the programmer selects any artifact derived from our explorer to record it into a journal, we can retrieve its origin by accessing the list of tracked messages.

In summary, this mechanism enables generic tracking of experiments and navigation steps through arbitrary exploratory tools. In our experiments, we did not experience any noticeable impact on runtime performance. While we have to exclude methods that use any of the aforementioned special bytecode instructions from tracking, we did not observe a significantly reduced recall, since modular tool extensions typically use accessor methods instead of reading instance variables directly and favor composition over inheritance.

References

[1] Maurice Amsellem. 1995. ChyPro: A Hypermedia Programming Environment for Smalltalk-80. In *Object-Oriented Programming, 9th European Conference* (Åarhus, Denmark) *(ECOOP '95)*, Mario Tokoro and Remo Pareschi (Eds.). Springer, Berlin, Heidelberg, 449–470.

[2] Kent Beck. 2000. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, USA. ISBN 978-0-201-61641-5.

[3] Kent Beck and Ward Cunningham. 1987. Expanding the Role of Tools in a Literate Programming Environment. In *Proceedings of the First International Workshop on Computer-Aided Software Engineering* (Cambridge, MA, USA) *(CASE '87)*. ACM, New York, NY, USA. http://c2.com/doc/case87.html

[4] Tom Beckmann, Joana Bergsiek, Eva Krebs, Toni Mattis, Stefan Ramson, Martin C. Rinard, and Robert Hirschfeld. 2025. Probing the Design Space: Parallel Versions for Exploratory Programming. *The Art, Science, and Engineering of Programming* 10, 1 (feb 2025), 33 pages. doi:10.22152/programming-journal.org/2025/10/5

[5] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 308–318. doi:10.1145/1453101.1453146

[6] Alan F. Blackwell, Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Chris Roast, Chris Roe, Allan Wong, and Richard M. Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Cognitive Technology: Instruments of Mind* (Warwick, UK) *(CT 2001)*, Meurig Beynon, Chrystopher L. Nehaniv, and Kerstin Dautenhahn (Eds.). Springer, Berlin, Heidelberg, 325–341.

[7] Max Boksem and L. Thomas van Binsbergen. 2024. Bridging Incremental Programming and Complex Software Development Environments. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments* (Pasadena, CA, USA) *(PAINT '24)*, Tom Beckmann, Luke Church, Robert Hirschfeld, and Mauricio Verano Merino (Eds.). Association for Computing Machinery, New York, NY, USA, 29–40. doi:10.1145/3689488.3689991

[8] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code Bubbles: A Working Set-based Interface for Code Understanding And Maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, GA, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 2503–2512. doi:10.1145/1753326.1753706

[9] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. 1998. Wrappers To The Rescue. In *Object-Oriented Programming* (Brussels, Belgium) *(ECOOP '98, Vol. 1445)*. Springer, Berlin, Heidelberg, 396–417. doi:10.1007/BFb0054101

[10] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. 2015. The Moldable Inspector. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Pittsburgh, PA, USA) *(Onward! 2015)*. Association for Computing Machinery, New York, NY, USA, 44–60. doi:10.1145/2814228.2814234

[11] Mihaly Csikszentmihalyi. 2008. *Flow: The Psychology of Optimal Experience* (1 ed.). Harper Perennial, New York, USA. ISBN 978-0061339202.

[12] Allen Cypher, Daniel Conrad Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky. 1993. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, USA.

[13] Richard Daft and Robert Lengel. 1986. Organizational Information Requirements, Media Richness and Structural Design. *Management Science* 32 (may 1986), 554–571. doi:10.1287/mnsc.32.5.554

[14] Alan R. Dennis, Robert M. Fuller, and Joseph S. Valacich. 2008. Media, Tasks, and Communication Processes: A Theory of Media Synchronicity. *MIS Quarterly* 32, 3 (2008), 575–600. doi:10.2307/25148857

[15] Diana DeStefano and Jo-Anne LeFevre. 2007. Cognitive Load in Hypertext Reading: A Review. *Computers in Human Behavior* 23, 3 (2007), 1616–1641. doi:10.1016/j.chb.2005.08.012

[16] Stéphane Ducasse. 1999. Evaluating Message Passing Control Techniques in Smalltalk. *JOOP: The Journal of Object-Oriented Programming* (jun 1999), 14 pages. https://hal.science/hal-05022582

[17] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. 2019. On the Effect of Discussions on Pull Request Decisions. In *Proceedings of the 18th BElgian-NEtherlands software eVOLution workshop (BENEVOL)* (Brussels, Belgium) *(CEUR Workshop Proceedings, Vol. 2605)*. https://ceur-ws.org/Vol-2605/16.pdf

[18] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland, UK) *(CHI '19)*. ACM, New York, NY, USA, 1–12. doi:10.1145/3290605.3300500

[19] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself, In Proceedings of the 12th Conference on Object-Oriented Programming, Systems, Languages, and Applications (Atlanta, GA, USA). *SIGPLAN Notices* 32, 10, 318–326.

doi:10.1145/263700.263754
[20] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. 2008. The Lively Kernel: A Self-Supporting System on a Web Page. In *Self-Sustaining Systems* (Potsdam, Germany) *(Lecture Notes in Computer Science)*, Robert Hirschfeld and Kim Rose (Eds.). Springer, Berlin, Heidelberg, 31–50. doi:10.1007/978-3-540-89275-5_2
[21] Mik Kersten and Gail C. Murphy. 2006. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Portland, OR, USA) *(SIGSOFT '06/FSE-14)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/1181775.1181777
[22] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, CO, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1265–1276. doi:10.1145/3025453.3025626
[23] Mary Beth Kery and Brad A. Myers. 2017. Exploring Exploratory Programming. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Raleigh, NC, USA, 25–29. doi:10.1109/VLHCC.2017.8103446
[24] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal, QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/3173574.3173748
[25] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (jan 1984), 97–111. doi:10.1093/comjnl/27.2.97
[26] Jens Lincke, Robert Hirschfeld, Michael Rüger, and Maic Masuch. 2008. SophieScript - Active Content in Multimedia Documents. In *Sixth International Conference on Creating, Connecting and Collaborating through Computing (CS '08)*. IEEE Computer Society, USA, 21–28. doi:10.1109/C5.2008.12
[27] Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a Live Development Experience for Web-Components. In *Proceedings of the 3rd SIGPLAN International Workshop on Programming Experience* (Vancouver, BC, Canada) *(PX/17.2)*, Luke Church, Richard P. Gabriel, Robert Hirschfeld, and Hidehiko Masuhara (Eds.). Association for Computing Machinery, New York, NY, USA, 28–35. doi:10.1145/3167109
[28] Mariam El Mezouar, Daniel Alencar da Costa, Daniel M. German, and Ying Zou. 2022. Exploring the Use of Chatrooms by Developers: An Empirical Study on Slack and Gitter. *IEEE Transactions on Software Engineering* 48, 10 (2022), 3988–4001. doi:10.1109/TSE.2021.3109617
[29] Roberto Minelli, Lorenzo Baracchi, Andrea Mocci, and Michele Lanza. 2014. Visual Storytelling of Development Sessions. In *International Conference on Software Maintenance and Evolution (ICSME '14)*. IEEE Computer Society, USA, 416–420. doi:10.1109/ICSME.2014.65
[30] Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2013. Expositor: Scriptable Time-Travel Debugging with First-Class Traces. In *35th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, San Francisco, CA, USA, 352–361. doi:10.1109/ICSE.2013.6606581
[31] Vreda Pieterse, Derrick G. Kourie, and Andrew Boake. 2004. A Case for Contemporary Literate Programming. In *Proceedings of the 2004 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries* (Stellenbosch, Western Cape, South Africa) *(SAICSIT)*. South African Institute for Computer Scientists and Information Technologists, ZAF, 2–9.
[32] Norman Ramsey. 1994. Literate Programming Simplified. *IEEE Software* 11, 5 (1994), 97–105. doi:10.1109/52.311070
[33] Patrick Rein, Stefan Ramson, Tom Beckmann, and Robert Hirschfeld. 2025. An Information Foraging Interpretation of Liveness. In *Symposium on Visual Languages and Human-Centric Computing* (Raleigh, NC, USA) *(VL/HCC)*. IEEE Computer Society, Los Alamitos, CA, USA, 128–138. doi:10.1109/VL-HCC65237.2025.00022
[34] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *The Art, Science, and Engineering of Programming* 3, 1 (jul 2019), 33 pages. doi:10.22152/programming-journal.org/2019/3/1
[35] Geoffrey Rockwell and Stéfan Sinclair. 2016. *Hermeneutica: Computer-Assisted Interpretation in the Humanities*. The MIT Press, Cambridge, MA, USA. doi:10.7551/mitpress/9522.001.0001
[36] David Röthlisberger. 2010. *Augmenting IDEs with Runtime Information for Software Maintenance*. Ph. D. Dissertation. University of Bern, Bern, Switzerland. https://boristheses.unibe.ch/1026/
[37] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. ACM, New York, NY, USA, 1–12. doi:10.1145/3173574.3173606
[38] David W. Sandberg. 1988. Smalltalk and Exploratory Programming. *SIGPLAN Notices* 23, 10 (1988), 85–92. doi:10.1145/51607.51614
[39] Ben Shneiderman. 1981. Direct Manipulation: A Step beyond Programming Languages. *SIGSOC Bulletin* 13, 2–3, 143. doi:10.1145/1015579.810991
[40] Jeremy Singer. 2020. Notes on Notebooks: Is Jupyter the Bringer of Jollity?. In *Proceedings of the 2020 SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Virtual, USA) *(Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 180–186. doi:10.1145/3426428.3426924
[41] Matthew Sotoudeh. 2025. Literate Tracing. In *Proceedings of the 2025 SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Singapore, Singapore) *(Onward! '25)*. Association for Computing Machinery, New York, NY, USA, 143–160. doi:10.1145/3759429.3762626
[42] Sakutaro Sugiyama, Takashi Kobayashi, Kazumasa Shimari, and Takashi Ishio. 2022. JISDLab: A Web-based Interactive Literate Debugging Environment. In *International Conference on Software Analysis, Evolution and Reengineering* (Honolulu, HI, USA) *(SANER 2022)*. IEEE Computer Society, 497–501. doi:10.1109/SANER53432.2022.00067
[43] Marcel Taeumel and Robert Hirschfeld. 2016. Evolving User Interfaces From Within Self-supporting Programming Environments: Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs. In *Proceedings of the Programming Experience 2016 Workshop* (Rome, Italy) *(PX/16)*. Association for Computing Machinery, New York, NY, USA, 43–59. doi:10.1145/2984380.2984386
[44] Marcel Taeumel and Robert Hirschfeld. 2021. Exploring Modal Locking in Window Manipulation: Why Programmers Should Stash, Duplicate, Split, and Link Composite Views. In *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming* (Cambridge, United Kingdom) *(Programming '21)*, Luke Church, Shigeru Chiba, and Elisa Gonzalez Boix (Eds.). Association for Computing Machinery, New York, NY, USA, 14–20. doi:10.1145/3464432.3464433
[45] Marcel Taeumel, Jens Lincke, Patrick Rein, and Robert Hirschfeld. 2022. A Pattern Language of an Exploratory Programming Workspace. In *Design Thinking Research: Achieving Real Innovation*, Christoph Meinel and Larry Leifer (Eds.). Springer, Cham, 111–145. doi:10.1007/978-3-031-09297-8_7
[46] Steven L. Tanimoto. 2023. Five Futures with AI Coding Agents. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming* (Tokyo, Japan) *(Programming '23)*, Shigeru Chiba, Youyou Cong, and Elisa Gonzalez Boix (Eds.). Association for Computing Machinery, New York, NY, USA, 32–38. doi:10.1145/3594671.3594685
[47] Christoph Thiede. 2024. The Semantic Workspace: Augmenting Exploratory Programming with Integrated Generative AI Tools. https://github.com/LinqLover/semexp-thesis/releases/download/submission/semexp-thesis.pdf Master's thesis.
[48] Christoph Thiede and Patrick Rein. 2023. *Squeak by Example* (6.0 ed.). Lulu. ISBN 978-1-4476-2948-1.
[49] Christoph Thiede, Marcel Taeumel, Lukas Böhme, and Robert Hirschfeld. 2024. Talking to Objects in Natural Language: Toward Semantic Tools for Exploratory Programming. In *Proceedings of the 2024 SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Pasadena, CA, USA) *(Onward! '24)*. Association for Computing Machinery, New York, NY, USA, 68–84. doi:10.1145/3689492.3690049
[50] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. Time-Awareness in Object Exploration Tools: Toward In Situ Omniscient Debugging. In *Proceedings of SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Cascais, Portugal) *(Onward!)*, Tijs van der Storm and Robert Hirschfeld (Eds.). Association for Computing Machinery, New York, NY, USA, 89–102. doi:10.1145/3622758.3622892
[51] David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the Experience of Immediacy. *Commun. ACM* 40, 4 (apr 1997), 38–43. doi:10.1145/248448.248457
[52] Tom Van Cutsem and Mark S. Miller. 2013. Trustworthy Proxies. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer, Berlin, Heidelberg, 154–178. doi:10.1007/978-3-642-39038-8_7
[53] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How Data Scientists Use Computational Notebooks for Real-Time Collaboration. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW, Article 39 (nov 2019), 30 pages. doi:10.1145/3359141
[54] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. 2000. Strengthening The Case for Pair Programming. *IEEE Software* 17, 4 (2000), 19–25. doi:10.1109/52.854064
[55] Paul Wuilmart, Emma Söderberg, and Martin Höst. 2023. Programmer Stories, Stories for Programmers: Exploring Storytelling in Software Development. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming* (Tokyo, Japan) *(Programming '23)*, Shigeru Chiba, Youyou Cong, and Elisa Gonzalez Boix (Eds.). Association for Computing Machinery, New York, NY, USA, 68–75. doi:10.1145/3594671.3594677
[56] Takashi Yamamiya, Alessandro Warth, and Ted Kaehler. 2009. Active Essays on the Web. In *Seventh International Conference on Creating, Connecting and Collaborating through Computing (CS '09)*. IEEE Computer Society, Los Alamitos, CA, USA, 3–10. doi:10.1109/C5.2009.10

Received 2026-01-09; accepted 2026-01-30