

Time-Awareness in Object Exploration Tools

Toward In Situ Omniscient Debugging

Christoph Thiede

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

Marcel Taeumel

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

Robert Hirschfeld

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

Abstract

Exploration of state and behavior is essential for understanding and debugging object-oriented programs. Many time-related questions about object communication – an object’s history – only arise in the context of a specific error in the here and now. At such a specific point in time, however, it is often distracting to involve omniscient debugging tools such as program tracers, because they do not integrate well with the programmer’s current focus on space-related questions and the informational cues at hand. In this paper, we present a novel way to provide a tangible, consolidated notion of both space and time in object exploration tools to make it more likely that programmers will use the available means to explore the evolution of particular objects. With programmers remaining informed about and in control of a program’s space and time, we promote the scientific method for debugging and leverage exploratory programming practices. We evaluate our model with hands-on experiences in the Squeak/Smalltalk programming system, using a program tracer that we have integrated into existing exploration tools to promote both spatial and temporal views. We believe that a clear, tangible notion of spacetime can help tool designers provide a better programming experience for those constantly recurring “What happened to this object?” situations.

CCS Concepts: • Software and its engineering → Software testing and debugging; Integrated and visual development environments.

Keywords: object-oriented debugging, omniscient debugging, object inspection, query-based debugging, program tracing, program exploration, exploratory programming, program tracing, moldable development, Smalltalk

ACM Reference Format:

Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. Time-Awareness in Object Exploration Tools: Toward In Situ Omniscient Debugging. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! ’23)*, October 25–27, 2023, Cascais, Portugal. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3622758.3622892>

1 Introduction

Programs run forward, producing a series of meaningful results. Account bookings get processed, weather simulations tick, pixels in a graphics scene are computed. In object-oriented designs, such domain concepts are realized through objects exchanging messages. Every so often, errors disrupt the flow of program execution, bringing everything to a halt and leaving programmers to explore, understand, and fix the situation. Tools for state inspection and behavior debugging help unveil erroneous paths; repeated or stepwise execution seems to “bend time” in the programmer’s favor. Eventually, program execution can go on normally and as designed. Thus, at best, programmers wish to remain informed about and in control of a program’s *space and time*.

The field of omniscient debugging investigates means to efficiently explore program execution without having to plan ahead (of time). Program tracers can take care of logging everything to enable “time-travel”, which means going back and forth to get an understanding of why something (un)expected has happened. In object-oriented programs, this manifests as programmers being able to explore the history of any object’s state and communication patterns in call trees. In contrast to (manually) repeated execution and thoughtful placement of breakpoints or logging statements, programmers can focus on the information at hand and navigate efficiently as questions arise.

However, existing approaches for time-travel tools dictate a shift in perspective without sufficiently integrating with the traditional, well-known tools for object exploration and (forward) debugging. They often expect an explicit commitment, which unnecessarily restricts the kind of questions that may arise. That is, one has to leave the current tool to consciously start the time-travel debugger or the tracer, to then ask questions in the realm of time or traces, like programmers having to decide between “release build” and “debug build” – as if



This work is licensed under a Creative Commons Attribution 4.0 International License.

Onward! ’23, October 25–27, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0388-1/23/10.

<https://doi.org/10.1145/3622758.3622892>

they could always know this in advance. It is challenging for programmers to manually combine answers to *space-related and time-related questions* to eventually understand the evolution of particular objects in the program.

Being experienced in malleable programming systems such as Squeak/Smalltalk, we studied the benefits of “debug mode is the only mode”¹ and thus the need for not having to plan ahead in terms of tool usage. When errors occur unexpectedly, the current object state leaves programmers puzzled, or stepwise (forward) execution only makes it worse, only then might time-related questions arise and beg for in-place tool support without losing the current focus.

This observation leads to the research question of this paper:

How can we design tools for program exploration that support both space-related and time-related questions and thus combine historical information about program execution (and object evolution) in a single workflow?

Thus, we want to further leverage the notion of exploratory programming [29, 33], where program design unfolds through onward experimentation and trial-and-error and cognitive effort is reduced by not having to plan ahead too far in the current task. Additional information can be queried as needed while staying focused on the data that is already visible and helpful.

In this paper, we make the following contributions:

1. We describe a practical approach for establishing a *universal tracing mode* (that records historic states) in exploratory programming systems while maintaining suitable performance for an immediate programming experience.
2. We present *spacetime exploration*, a novel interaction model for object inspection and debugging tools that enables programmers to explore objects along the two dimensions of space and time on par.
3. We discuss the feasibility of the model by applying it to our prototypical implementation of the TRACEDeBUGGER² and sketching how other existing toolsets can be subsumed within our model.

We believe that tool designers can use our insights to integrate time-travel mechanisms into their environment to be used more often by programmers who struggle with time-related questions but hesitate to start over their current debugging session just to fire up that program tracer successfully.

¹Gilad Bracha. 2012-11-17. *Debug Mode is the Only Mode*. <https://gbracha.blogspot.com/2012/11/debug-mode-is-only-mode.html> Accessed: 2023-04-28.

²<https://github.com/hpi-swa-lab/squeak-tracedebugger>

In [section 2](#), we provide background information on how we think about programming, tools for exploration and debugging, and the integration of tools in a programming environment. In [section 3](#), we describe how programming systems can trace exploratory activities continuously with a reasonable performance. In [section 4](#), we present our model for time-aware object inspection tools which promotes traditional exploration practices and ad-hoc support for time-related program comprehension questions. In [section 5](#), we apply our tool model in Squeak/Smalltalk, including two examples of how the spacetime explorer could manifest for domain-specific tools, including an evaluation of the system performance in [section 6](#). We discuss the results in [section 7](#) and conclude our thoughts in [section 8](#).

2 Background and Motivation

In this section, we explain our notion of exploratory programming and how debugging is limited in existing systems that support techniques for exploration and experimentation. In particular, we will argue using a vocabulary around object-oriented programming and Smalltalk programming systems.

2.1 Exploratory Programming

Exploratory programming is a programming technique for working on a software system where the system or the requirements are not fully understood [3, 29, 33]. It allows programmers to iteratively refine their knowledge of the system and the problem space and to prototype and evaluate possible solutions.

During exploratory programming, programmers conduct many experiments by having a conversation with the system in which they ask questions and interact with parts of the system to form, confirm, and reject hypotheses [28, 37]. These questions can relate to the *space* and to the *time* of the system:

Space-related questions refer to the *state* of the system, i.e., the meaning of the data stored in the system, its structure, and its interconnections.

Time-related questions refer to the running *behavior* of the system, i.e., the way the system creates, uses, modifies, or arranges data.

Typical exploratory systems allow programmers to ask questions by selecting items from a list or by evaluating short code expressions as *programs* in the context of the system.

Exploratory programming thrives through low-cost experiments where programmers are encouraged to ask a large number of questions [33]. Thus, by avoiding interruptions to programmers that are caused by temporal delays or levels of indirection [29], supportive programming systems should provide an experience of *immediacy* [41]:

Temporal immediacy: “Human beings recognize causality without conscious effort only when the time between causally related events is kept to a minimum.”

Spatial immediacy: “[...] means the physical distance between causally related events is kept to a minimum.”

Semantic immediacy: “[...] means the conceptual distance between semantically related pieces of information is kept to a minimum.”

The interactive programming system *Smalltalk-80* is founded on exploratory programming practices [9, 33]. Being purely object-oriented, all actions are modeled through messages exchanged between objects. Every object is an instance of a class and is characterized by its *identity* which distinguishes it from all other objects in the system, its *state* which is represented by its primitive slots or variables, and its *behavior* which is described by methods that implement the reception of messages. For example, programmers can model a database as a set of data structure objects whose state includes their data, metadata, and caches, and whose behavior includes their ability to construct themselves or to manipulate and access data.

Squeak/Smalltalk is a modern implementation of such a completely *open* and *explorable* programming system: programmers do not think of isolated applications but of interacting systems, being able to access and manipulate all objects and processes in it. For that, the system provides them with tools to explore objects by asking them questions about their state and behavior. Squeak is implemented in itself [13] and allows programmers to explore and manipulate the entire fundament on the fly, including its compiler, interpreter, and tools for browsing and debugging [12, 32, 38].

Next to Smalltalk programming systems, two other common examples of interfaces with a solid experience of immediacy are REPL interpreters and computational notebooks [42], both of which also support strategies for exploratory programming.

2.2 Tools for Program Exploration

Programmers often need to explore the system at hand for various types of tasks such as prototyping a feature, discovering an extension point, or fixing a bug. For example, a standard library programmer might want to optimize write access for the `RunArray` class, which is a dynamic sparse collection in Squeak, and thus explore how actual `RunArray` objects implement the addition of new elements. For such an exploration task, exploratory programming systems provide two types of tools: object inspectors and (process) debuggers.

Object inspection. Programmers can explore the *space* of a system by inspecting objects by their state. For this, they can use *general-purpose* tools that are provided by the programming system or *domain-specific* tools from the system under exploration. For example, a popular general-purpose

inspection tool is a *property sheet*, which displays the individual slots and variables of an object in a key-value table [18; 38, chap. 6, sec. 3]. Domain-specific tools can provide higher-level representations of an object such as a directed graph for a complex data structure, a visualization of statistical data, or a preview of graphical resources [4].

Object inspection tools are typically *interactive*, allowing programmers to ask more specific questions by selecting portions of the object state to request more details about it and thus navigate through the object graph. For example, our standard library programmer can inspect a sample `RunArray` object through a general property sheet and select its `runs` variable to examine its assigned integer array more closely and learn about its semantics, or she can evaluate a custom query to access or count elements in the `RunArray`.

(Process) debugging. Programmers can explore the *time* of a system through the behavior of objects. They can learn about the behavior by *browsing* the object’s implementation, but due to the abstract nature of static source code, this activity often involves additional complexity and a limited experience of immediacy. Instead, many exploratory programming systems promote *dynamic* program execution where programmers can explore behavior in the context of a concrete program instance.

A common dynamic program exploration tool is the *debugger*, which allows programmers to interactively execute a program step-by-step while having access to the *context stack* of currently active methods, their source code, and the state of the relevant objects [18; 38, chap. 6, sec. 4]. *Omniscient debuggers* (also *back-in-time debuggers* or *time-travel debuggers*) enhance the immediacy of debugging by enabling programmers to freely navigate through a *context tree* (also referred to as *call tree*) that contains all method activations over time [11, 17, 27]. For that, they either record a program trace with the context tree and, optionally, snapshots of the historic states in advance [20], or they re-run a reproducible program to interactively display the information requested by the programmer [23]. In addition to the order of execution, omniscient debuggers can provide further means of navigation, including queries about the dataflow of information (i.e., *dynamic slicing*) [14, 19, 40], the global object graph [10, 15, 26], changes to the state of objects over time [7, 8, 34], and events in the program execution [5, 6, 16, 25, 31].

In our example of a standard library, our library developer can debug the enumeration of the `RunArray` object to investigate how its elements are accessed in order. Alternatively, she can back-in-time debug the construction of a new instance to retrace the initialization of its internal variables. Both strategies can help gain a better understanding of the situation.

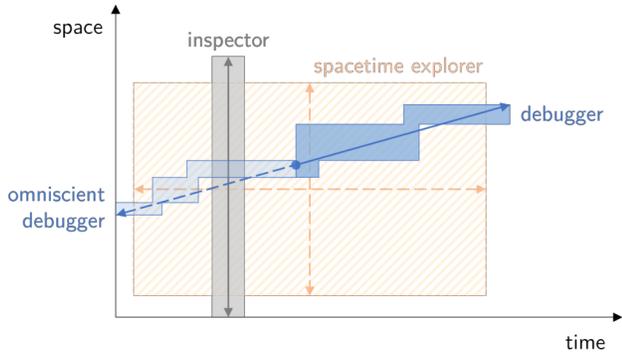


Figure 1. Programming tools help explore a system’s *space* (or state) over its *time* (or behavior). Traditional tools (here: inspector and debugger) provide limited means for navigating the dimensions of space and time. Our *spacetime explorer* offers more flexibility for richer support of program-comprehension questions.

2.3 Toward Immediacy Across Space and Time

Object inspection and process debugging are often interwoven as most programmers’ exploratory journeys take them through both space and time (fig. 1). Yet, typical exploratory programming systems provide separate means for each activity, and programmers frequently alternate between them. For example, our library programmer starts by inspecting a sample `RunArray` object to find an interesting variable in its property sheet. She then closes the inspector and switches to a debugger to explore the emergence of the variable’s value during the array’s initialization. As she steps through the execution, she inadvertently overshoots and has to backtrack a couple of times. Finally, she finds a valid explanation for the value, which means closing the debugger and switching back to the original inspector. Another hunch leads her to search for another variable, which entails inspecting the properties of that variable’s value in another inspector. The exploratory journey continues.

We envision a new kind of programming tools that subsume the navigational axes of space and time in a generally explorable way. Such tools should ensure the omnipresence of time- and space-related information. Programmers should not be forced to plan ahead but rely on being able to (1) explore the system state at all abstraction levels and (2) freely go back and forth to learn about the evolution of that state over time. In particular, we want to tackle the following challenges:

Temporal immediacy: Due to the nature of program execution, moving forward in time is possible by advancing the program, but if programmers want to move backward, a trace of the prior execution is required. Since program tracing involves a significant

performance overhead, programming systems usually do not allow for going back by default.

Spatial immediacy: Poorly integrated tools for exploration exhibit spatial clutter and rely on programmers to manually (or mentally) connect useful pieces of information. Redundant switches between multiple views and perspectives increase the cognitive load and thus the chance of making mistakes.

Semantic immediacy: A cluttered toolset reduces the effective amount of useful information because being side-by-side does not imply being related. Many inspection tools do not provide programmers with related information about the emergence over time of the state being viewed; many debuggers offer only a limited perspective on the state of objects from the current context frame.

We believe that by establishing a *universal tracing mode* and a flexible *spacetime exploration model*, tools for exploratory programming can further improve the overall programming experience while designing and implementing high-quality software.

3 Establishing a Universal Tracing Mode

Exploratory programming systems usually do not make historic information about program execution available on the fly. The typical resource requirements of program tracers conflict with the need for short feedback cycles and an experience of temporal immediacy. Thus, the prospect of increased execution time and memory usage demands a trade-off to at least get the feeling of having *in-situ omniscient debugging*.

In this section, we describe our ideas for establishing a universal, omnipresent, “feels like always-on” tracing mode in exploratory programming systems without sacrificing too much temporal immediacy.

3.1 Efficient Trace Model

A naive approach to representing the historic data of a program trace is to store a full snapshot of all objects at each point in time. Yet, this is a highly redundant approach and introduces a significant memory/storage overhead since many program instructions will not change any or only a few variables of the overall system state. Specifically in exploratory programming systems, many programs evaluate programmer-initiated questions (or queries) that only regard one or a few sub-systems; these (mini-)programs will most likely not manipulate a larger fraction of objects in the system.

In our program tracer [39], an *incremental historic memory* efficiently stores fine-grained changes in a sparse collection for each object and each of its slots (i.e., variables) using a hash table (fig. 2). When active, the tracer detects side effects to object slots and stores previous values in the historic memory before they are displaced. We avoid duplicating current

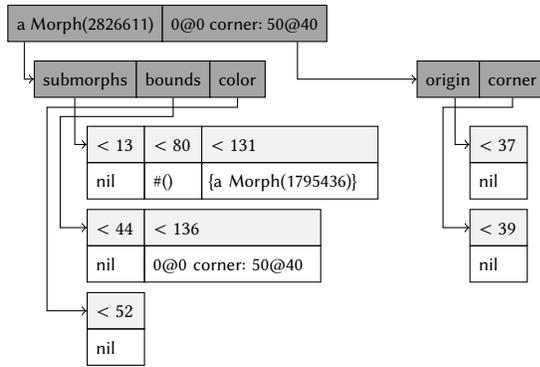


Figure 2. Example of our incremental historic memory structure for tracing the construction of a widget (Morph) in Squeak. The dark boxes represent associative arrays of pointers. For each object slot involved, all former displaced states are preserved in a sparse array. For example, the morph’s submorphs variable was assigned the empty collection #() during the time interval [13, 80).

values in the history since these remain available in the running system. Once the traced program has terminated, the current system state is not discarded until the “host system” (or the entire Squeak/Smalltalk environment in our case) terminates. Consequently, the size of the memory efficiently grows with the number of performed side effects, making it more or less independent of the program’s execution time or the system’s size. For the same reasons, the time to write to the historic memory is reduced as well.

3.2 Explicit Exploratory Interface

A strength of exploratory programming systems is that programmers can explore, modify, and interact with any part of the running system at any time. Yet, this poses a challenge to program tracers which are intended to record programmer-initiated behavior only. For example, a typical user interface widget in Squeak continuously receives drawing messages from the Morphic rendering engine, but programmers can also directly interact with the widget to change its appearance or trigger actions. In this situation, the programming system is unable to distinguish between the “background noise” of the rendering system and the behavior that happens in response to the programmer’s interactions.

To allow for this distinction, we define *explicit exploratory interfaces* in the system and enable the program tracer whenever the execution crosses the boundaries of these interfaces. For example, we define the Squeak compiler’s protocol for evaluating custom expressions as an exploratory interface. Analogously, we can define the means of interaction through inspection tools or the Morphic halos for direct manipulation [38, chap. 12] as exploratory interfaces. Thus, programmers can execute the system and interact with its part as

usual, and the system will trace small portions of the running behavior only, keeping the performance overhead low and preserving the experience of temporal immediacy.

As an extension to breakpoint debugging and in the case of unhandled exceptions, explicit exploratory interfaces can help create a sense of in-situ omniscient debugging. Given a traditional debugger that displays a suspended process and thus a specific point in execution time, selected method activations (or stack frames) might be eligible entry points for *program reproduction* if the affected state can be rewound. For performance reasons, promising candidates can typically be found at the time of object creation, when the entire context is sufficiently defined. Note that techniques around exploratory programming can cope with “good enough” situations as long as programmers can reason about the information at hand and thus notice irregularities to avoid false conclusions. Consequently, *ad-hoc* back-in-time navigation might not work for all unplanned debugging situations but it might be worth a try.

3.3 Program Reproduction

As an alternative to tracing program execution in advance, programmers can also re-run programs multiple times to collect relevant information [24] as needed. To reproduce the execution, still, they need to provide the original entry point again. This can be challenging as the state of an object might have changed through a series of interactions through different tools and interfaces.

We suggest that exploratory systems can log these interactions through explicit exploratory interfaces (as described above) to automatically reproduce them later for tracing. In particular, several exploratory systems already maintain such interaction logs that could be exploited: for example, Squeak records all evaluated expressions in a *changes file* [38, chap. 7, sec. 4] and many REPL interpreters maintain a history of previously executed commands.

However, reproducing program execution in combination with re-tracing parts of the call tree requires *deterministic* state changes. Without explicit guards or isolation, a program’s side effects might even affect other activities in the system. We therefore suggest that a universal tracing mode should prioritize upfront tracing and employ program reproduction only sparingly. In ambiguous situations, the resulting trace should be flagged so that programmers can make their own educated guesses and decide how to proceed. Note that programmers even anticipate such situations on their exploratory journey.

4 The Spacetime Exploration Model

To enable programmers to handle and exploit the temporal dimension of program state, we propose *spacetime exploration*, a new interaction model for exploratory programming tools that combines object inspection and debugging into a single

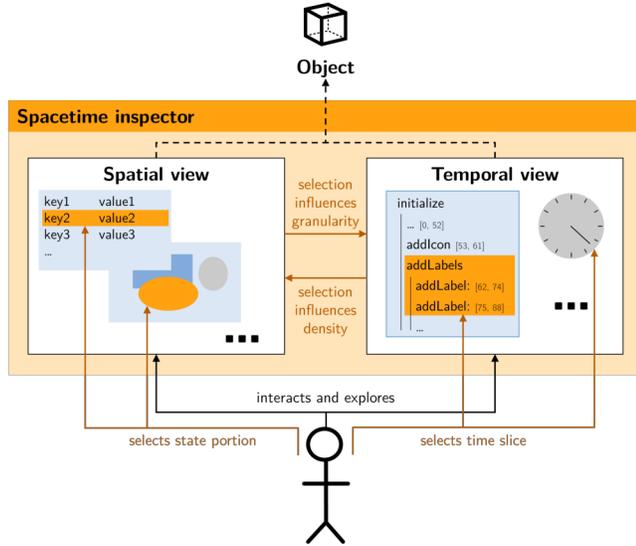


Figure 3. Our tool model for spacetime exploration. A programmer explores an object’s spatial and temporal dimensions (blue). The spacetime inspector combines views on both dimensions in a single workflow (orange). Traditional views include property sheets or trees; advanced visualizations can enrich the experience. Programmers retain control over the informational granularity of both dimensions.

workflow. In our model, programmers explore objects by two dimensions on par: space, through the object’s state, and time, through the object’s behavior. Both dimensions are tangible³ entities that programmers can directly interact with to navigate through the spacetime of the explored object.

The proposed interaction model consists of four central entities (fig. 3):

Object: The model focuses on a selected object from the system under exploration. It is part of an executed or executable program instance.

Spatial view: The spatial view provides access to the state of an object. The object state can be represented either through a general-purpose tool such as a property sheet or a hierarchical list for nested objects or through a domain-specific visualization. Usually, programmers can interact with the spatial view to refine it for specific questions.

Temporal view: The temporal view provides access to the history of an object. Time can be represented by a flat timeline or be structured according to the system’s behavior. Like the spatial view, the temporal view can be a general-purpose artifact such as a context tree or be tailored to the semantics of the specific object’s domain. By interacting with the temporal view, programmers can navigate through the object’s history.

³In Smalltalk systems, *tangibility* refers to the ability to send messages to the object at hand to explore it or trigger side effects.

Programmer: The programmer explores the object in the context of the program through the spatial and the temporal view. To navigate through the spacetime of the object, she can interact with both views to control the amount of detail in the spatial and temporal dimensions.

Both views influence each other dynamically: the space is displayed for the current selection of the time, whereas the time is displayed for the history of the selected space. So, by choosing a relevant *state portion* (i.e., a set of variables) in the spatial view, the programmer can adjust the granularity of the temporal view. Analogously, by selecting a *time slice* (i.e., an interval between two points in time) in the temporal view, she can adjust the density of information displayed in the spatial view.

For example, the programmer could explore an object from a traced program in a spacetime-aware inspector. Within the inspector, she could choose between different spatial views that provide different levels of informational density, such as a single variable view vs. a full property sheet. Based on the spatial view she chooses, the temporal view will display the time with a different granularity: for the single variable view, the relevant variable might have been reassigned only two times, so the temporal view would display only two method activations for these side effects. On the other hand, the property sheet displays multiple variables that correspond to a larger set of side effects, and the temporal view will display a larger number of method activations. Conversely, the spatial view displays a list of all historic values for each variable, corresponding to a high information density. Yet, if the programmer selects a subset of the method activations in the temporal view, the spatial view will display only those variable values that have been changed from one of the selected method activations.

Thus, the proposed interaction model builds on the concept of *object traces* which describe the history of all changes to a state portion of an object [39]. Yet, an object trace distinguishes between a query for selecting the state and a list for outputting the results, whereas the spacetime exploration model merges both of them into a single interactive view. Note that spacetime exploration is different from traditional *query-based debugging*, which requires programmers to provide explicit queries to retrieve values from both dimensions independently [8, 10, 25]. For spacetime exploration, they only express a simple query to request a state portion, and this query implicitly performs a drill-down on the time, or conversely, they only ask for a subset of the time to create an implicit filter on the space.

5 Applying Spacetime Exploration in Squeak/Smalltalk

We demonstrate the practicality of our proposed interaction model by describing a prototypical *spacetime inspector* in

Squeak/Smalltalk. We draw inspiration from similar concepts in other domains such as revision control systems [2, 30] (e.g., git⁴), versioning file systems [21], or the Internet Archive’s Wayback Machine⁵. For instance, git’s *log* and *blame* features enable programmers to explore the space (files and lines) and the time (change history) of a system (the repository) similar to our spacetime exploration model, and programmers can control the density/granularity of either dimension to reduce the complexity of the resulting artifacts.

5.1 Implementing the Spacetime Inspector for the TRACEDEBUGGER

The TRACEDEBUGGER is an omniscient debugging tool for Squeak/Smalltalk that allows programmers to incrementally execute a program and maintains a program trace for later exploration. For the program trace, the TRACEDEBUGGER records the behavior in a context tree and all states in an incremental historic memory. For accessing historic states, it provides a *range retracing* mechanism that evaluates a user-provided *range query* against the incremental historic memory and returns an *object trace* of all different results of the query over time [39]. Thus, the granularity of the object trace, i.e., the number of individual results, depends on the complexity of the state requested by the query as well as on the number of side effects that have been made to that state during the traced program execution. To implement tracing and retracing, the TRACEDEBUGGER instruments the Squeak bytecode interpreter for tracking single instructions and emulating historic states.

We design the spacetime inspector as a general-purpose tool that programmers can extend with domain-specific spatial views. Programmers can open an object in the spacetime inspector and dynamically switch between all supported spatial views. Thus, the spacetime inspector is a moldable tool [4, 5]. Predefined spatial views include a simple textual representation of the object and a hierarchical property sheet for exploring the state and composition of the object that resembles Squeak’s built-in object explorer [38, chap. 6, sec. 3]. Because programmers can select not only a single point in time but also a time slice in the temporal view, each spatial view can aggregate multiple versions of the object and display them side by side. In our prototype, the temporal view represents the behavior of the system through a slice of the original context tree, which is displayed in an interactive tree view where programmers can select any subtree to navigate through time. Additionally, a slider is provided to scrub through the individual points in the time slice, i.e., the leaves in the corresponding context tree slice.

We implement the spacetime inspector in the context of the TRACEDEBUGGER using object traces (fig. 4). For that, each spatial view provides a range query to the framework

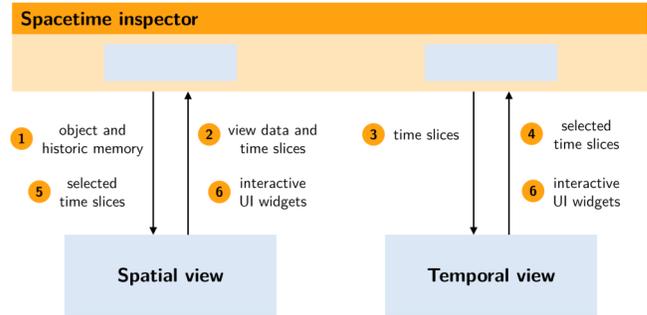


Figure 4. Expected dataflow between custom spatial/temporal views and the spacetime inspector. The spatial view retrieves an object and the historic memory (1) and provides a list of view data and time slices (2). The temporal view retrieves the time slices (3) and provides a list of selected time slices (4). Based on this selection, the spatial view retrieves the corresponding view data (5). Both views contribute to the spacetime inspector’s UI (6).

that retrieves a relevant state portion from the explored object, and the framework returns an object trace containing all query results within the user-selected time slice to the view. When programmers interact with a spatial view, it can update its query dynamically: for example, if a programmer expands the subtree for one value from a property sheet, the query is refined to collect all nested properties from the relevant object as well. As a result, the granularity of the resulting object trace might increase, causing the temporal view to display additional method activations in the tree view.

5.2 Use Case: Exploring Evolution of a RunArray

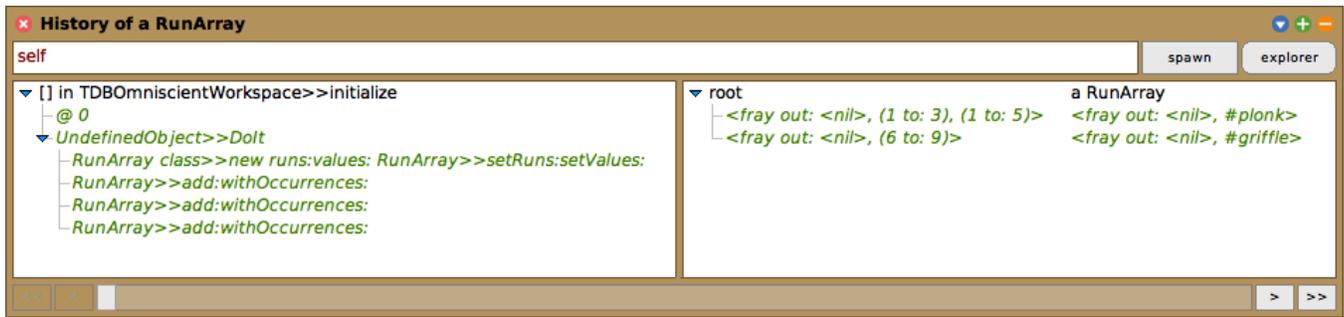
The RunArray is a run-length encoded data structure for dynamic sparse collections in Squeak. Internally, it maintains two arrays: runs, which contains the number of repetitions for each element, and values, which contains the pairwise different elements of the collection. When a new element is added to the collection, the RunArray first attempts to extend the last existing run by increasing the last item of the runs array, or otherwise, if the new element is different from the previous one, it updates both arrays by copying them into new larger arrays and appending the new value and run length to them.

We describe how our library programmer can use the spacetime inspector to explore the evolution of her sample RunArray object and get an understanding of its implementation. She creates the RunArray using the following code:

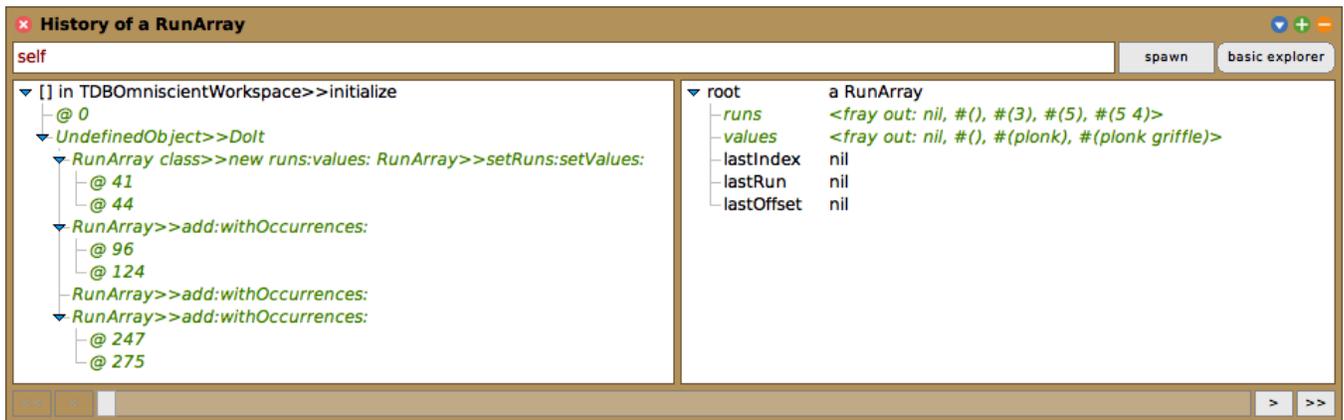
```
RunArray new
  add: #plonk withOccurrences: 3;
  add: #plonk withOccurrences: 2;
  add: #griffle withOccurrences: 4;
  yourself
```

⁴<https://git-scm.com>

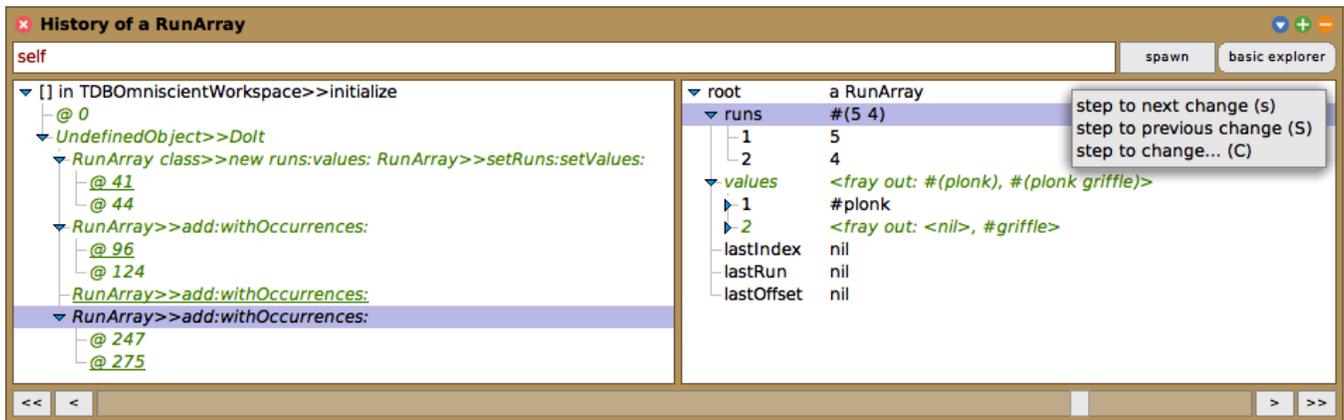
⁵<http://web.archive.org>



(a) Exploring the RunArray through a domain-specific spatial view that displays the single elements over time. Invalid intermediate states have been hidden through a filter.



(b) Exploring the RunArray through a generic property sheet that displays its basic variables over time. This screenshot is the result of clicking on the representation mode “explorer” at the top right corner of the window in fig. 5a and choosing “basic explorer” instead.



(c) Exploring the addition of an element to the RunArray through a generic property sheet with a reduced informational density. In the property sheet, the user has selected and expanded the runs variable to highlight all changes to this variable in the temporal view (underlined blame). Additionally, she can use the context menu of the variable to navigate to its adjacent changes.

Figure 5. Implementation of our spacetime exploration model for exploring the history of the RunArray object with different levels of detail. Note that we chose to place the temporal view on the left and the spatial view on the right. By dragging the horizontal slider, users can move fluidly through the points in time from the temporal view. *Inline summaries* (here: “fray out”) provide an overview for heterogeneous values.

She also wonders how the runs array has changed, i.e., whether the array object has been mutated inline or whether the variable has been assigned a copy of its previous value. To answer this question, she uses the *blame* feature of the spacetime inspector and selects the variable in the spatial view (fig. 5c). In response, the temporal view highlights all other method activations in the tree that have changed the runs variable itself (i.e., that reassigned it) and offers her a pair of buttons to navigate between these methods. Based on this information, the library programmer realizes that the runs variable has been reassigned many times, so she suspects that for each write operation, the RunArray performs an expensive copy operation, and she verifies this hypothesis by opening the addition method in a code browser. Finally, she decides to optimize the data structure by implementing an overallocation strategy in this method.

5.3 Use Case: Visualizing Regular Expression Matching

Here, we illustrate the versatility of the spacetime exploration model by using it to visualize the matching behavior of a regular expression engine. The Regex package in Squeak provides a Smalltalk-specific flavor of regular expressions. It implements pattern matching through a non-deterministic finite automaton (NFA) and object-oriented recursion: given an input string, the matcher recurses through the NFA while at each link trying to match the next character against predicates or contextual conditions and capturing matches. In our visualization, we want to explain the high-level concepts of the regular expression engine by exploring the matcher's behavior through real examples.

For this, we combine two spatial views and one domain-specific view (fig. 6). All views are domain-specific and correspond to our definition from section 4. The first spatial view is a text widget that displays the input of the matcher and highlights the range of the attempted match substring. The second spatial view is a graph widget that displays the structure of the matcher's NFA (which is constant per pattern) through a set of labeled nodes and directed edges⁶. The temporal view visualizes the behavior of the matcher: on top of the graph widget, we draw a spline curve that resembles a literal thread and displays the current recursion stack by winding through the links on the active recursion path.

Through the interface of the spacetime inspector, we can retrieve the required data for the views regardless of the specific memory data structure of the program tracer by expressing queries against the state of the matcher object such as self startPosition to: self position or:

```
ThisContext7 stack
  select: [:ctxt | ctxt selector = #matchAgainst:]
  thenCollect: [:ctxt | ctxt receiver]
```

⁶<https://github.com/LinqLover/Regex-Tools>

⁷ThisContext refers to the active context frame during the execution.

The spacetime inspector allows programmers to interact with the visualization. They can drag the time slider to observe the growth of different recursion paths and understand how the matcher backtracks. In addition, the spacetime inspector interface allows to easily define further means for interaction, such as navigating the time by selecting nodes in the tree, adjusting the granularity of the visualization by expanding or collapsing sets of nodes in the graph, or exploring the captured substrings through another spatial view that supports the blame feature.

6 Evaluation of Performance

We evaluate the performance of our proposed approaches for establishing a universal tracing mode and the spacetime exploration model in terms of their time and space consumption. Even though this is not a user study, we can assume that a good system performance has a positive effect on user performance and satisfaction [41].

6.1 Universal Tracing Mode: Responsive up to Medium-sized Workloads

For our current prototype, we sacrificed maximum performance for a simple implementation and built the program tracer through Squeak's image-side bytecode interpreter rather than using instrumentation or a virtual machine-side approach. Thus, compared to regular execution in the virtual machine, tracing introduces a runtime overhead of between 100 000 % and 1 000 000 %. Still, due to the explicit exploratory interfaces that we define in the programming system, we only need to trace short interactions of the programmer with the explored system, which introduces small delays of only a few seconds for small- to medium-sized workloads (table 1).

Table 1. Time and memory consumption of the TRACEDeBUGGER recording program traces for different workloads.

Domain	Program	Time [s] ^a	Memory [kB]
<i>Data structures</i>			
	RunArray new		
	add: #plonk withOccurrences: 3;		
	add: #plonk withOccurrences: 2;		
	add: #griffle withOccurrences: 4;		
	yourself	0.0021	61.4
<i>Regular expression matching</i>			
	"matcher := \"d+(\d+)*asRegex."		
	matcher matchesIn: '1		
	17_3 42_7895_0_456		
	999999_0_0_' readStream	0.407	11 782
<i>UI widget construction (13 elements)</i>			
	WatchMorph basicNew initialize	0.797	15 299
<i>UI rendering (89 elements, 650 px × 425 px)</i>			
	aSystemBrowserWindow imageForm	8.905	2 567 832

^a Test machine: Intel i7-8550U CPU @ 1.80 GHz. Environment: Open Smalltalk Cog/Spur VM of version 202206021410.

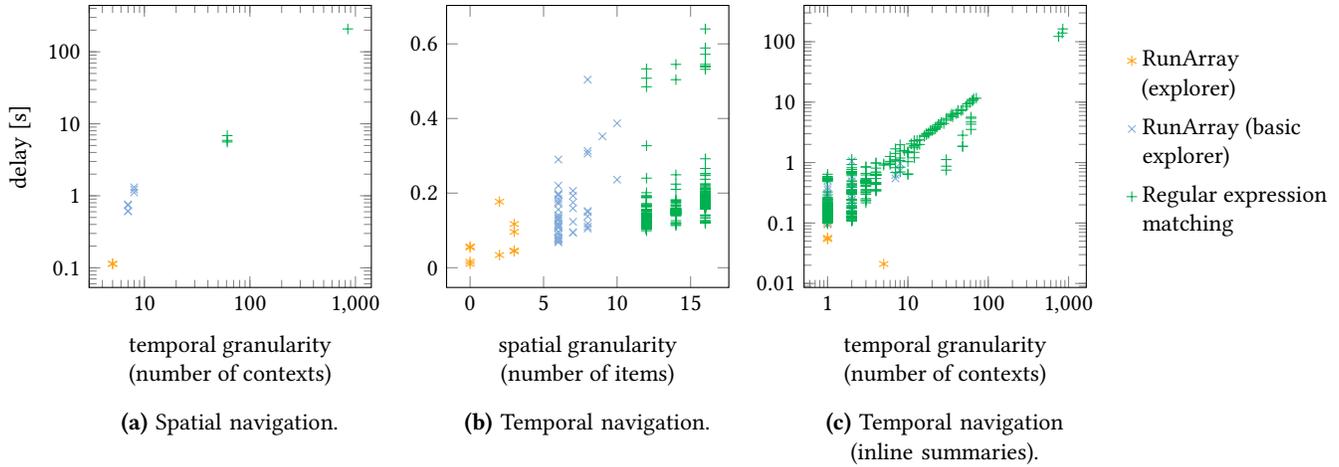


Figure 7. Interaction delays for navigating the spacetime inspector. Values were recorded for different workloads that were described in sections 5.2 and 5.3.

Moreover, the tracing costs are amortized over the single interaction steps performed by the programmer (e.g., through separately evaluated commands), resulting in interactive response times for most workloads [35, p. 473].

As a result of our space-efficient memory model, most of our considered program traces consume only a small number of megabytes (table 1). The only exception is a CPU-based rendering system running in unoptimized legacy mode⁸ whose trace consumes multiple gigabytes.

6.2 Spacetime Exploration: Temporal Granularity Matters

We show that spacetime exploration tools can be built with a practical performance by the example of the presented spacetime inspector. We measure the delays caused by user interactions for spatial navigation (i.e., changing the granularity of state) or temporal navigation (i.e., changing the selected time slice).

Delays for spatial navigation correlate with the temporal granularity of the spacetime inspector (number of contexts displayed in the tree), ranging from 0 to 2 seconds for small-sized workloads and rarely exceeding 8 seconds for larger workloads (fig. 7a). Delays for temporal navigation correlate with the spatial granularity (number of items displayed in the spatial view) and do not exceed 0.7 seconds for any of our observed workloads (fig. 7b). A bottleneck is inline summaries (“fray-outs”, fig. 5) which introduce delays of more than 1 second for medium-sized workloads up to several minutes for a high temporal granularity (fig. 7c). Note that inline summaries are an optional feature: they are only generated when selecting inner nodes in the context tree and users who only use the time slider for navigation will never encounter

⁸Currently, the TRACEDEBUGGER supports only the lowest-level rendering primitive of Squeak’s BitBlt plugin (primitiveCopyBits) and triggers the less efficient image-side fallback code for all other rendering primitives.

them. Thus, spatial navigation through an object’s history satisfies Shneiderman’s criteria for common interactive tasks for small- and most medium-sized workloads and temporal navigation meets the criteria for frequent interactive tasks for any small- to medium-sized workloads [35, p. 473].

Moreover, the spacetime inspector constitutes an application of spacetime exploration that is domain-agnostic and promotes object-centric, contextual information. For domain-specific views, programmers can exploit domain knowledge for optimizations, and spatial and temporal granularity tends to be smaller. For example, our visualization of regular expression matching (section 5.3) only requires two spatial items at a static temporal granularity, resulting in a query that we can evaluate once in less than 4 seconds and cache for all navigation steps.

7 Discussion and Related Work

In this section, we discuss the implications and limitations of our work for programmers and tool developers and compare it to related work.

7.1 Programming Experience

Our proposed interaction model streamlines program exploration by combining two related yet typically disconnected activities: object inspection and (omniscient) debugging. This promotes tools for exploratory programming where tool builders aim for focused, uninterrupted programming sessions, supporting a wider range of exploration tasks from a single place and thus contributing to the users’ experience of immediacy.

Further, tools can use the dimensions of space and time to provide exploratory programmers with richer contextual information about a system, assisting them to discover potentially relevant relationships between objects and events earlier. Through the omnipresent spatial history of objects under

exploration, programmers gain access to another source of information that can help them develop an intuitive understanding of the semantics of the state of complex objects and that is often easier to comprehend than the complex source code that produced the state in question.

Finally, the spacetime exploration model equips programmers with a more consolidated metaphor for reflecting on their interaction with the system and expressing clearer questions that relate to both the state and the time of a system. Programmers face a smaller gulf of execution [22] for communicating these questions to the system: they do not need to explicitly describe both dimensions separately as in existing approaches to queryable debugging, and views can encapsulate the actual query behind a higher-level (interactive) representation.

7.2 Building Tools for Spacetime Exploration

Our presented applications of the concept show that the interaction model can be practically implemented and used by programmers. In particular, the explicit notion of views in the spacetime inspector makes it possible to reuse and compose existing tools for representing and exploring the space or time of objects from arbitrary or specific domains. Still, we acknowledge the intrinsic complexity of tool building and the lack of support for moldable development in many environments that might discourage programmers from building domain-specific tools.

The design of the spacetime inspector does not require these tools to be aware of the spacetime inspector. In practice, still, they can benefit greatly from being tailored to the spacetime context: first, this allows them to display the relevant state of an object over an entire time slice by displaying changed parts in an aggregated summary. Second, tool builders can significantly improve their responsiveness by factoring out UI logic that is independent of object state from the range query.

7.3 Applicability and Limitations

Due to its object-centric nature, the applicability of the spacetime exploration model is limited to systems that model relevant state portions in a coherent region of the object graph across time. For example, programmers might gain less insight into a system that models state changes in a functional style through immutable copies of objects rather than side effects, has weak semantics of object identity (e.g., due to proxies or serialization concerns), or entails object-crossing dataflow where values are moved and rearranged throughout different regions of the object graph. Since dataflow in particular is common in many systems, spacetime exploration is not a suitable means in general for retracing the infection chain of a bug [23]. Instead, we see the model's greatest potential in exploratory tasks that center on the evolution or emergence of an object or a composition of objects, allowing

programmers to explore the operating principles of systems through domain-specific views on their state and behavior.

The spacetime model necessarily depends on a resource for exploring time that is typically expensive to obtain, store, and query, since program tracing and object traces are often connected to high CPU and memory consumption [20, 39], and in many systems, programmers might not even have access to a program trace. We can mitigate these concerns by tracing only relevant subsets of the program execution or by lazily reconstructing and rerunning programs; note that the latter approach might be unreliable for programs that depend on global state or non-deterministic behavior. Due to the dynamic structure of a spacetime inspector, its optimization potential through caching and pre-filtering is limited. These performance limitations can limit the responsiveness of a spacetime inspector depending on the complexity of the system and the views and thus reduce the immediacy of spacetime exploration; fortunately, we have found that our prototype implementation is already sufficiently responsive for many practical small- to medium-sized workloads.

7.4 Related Work

Similar to our proposed spacetime exploration model, [1] describes another approach to combine the dimensions of space and time of a program trace by displaying them in a trace table. In contrast to our object-centric perspective, their model centers on a stackframe or subtree of the call tree and examines the dataflow of values rather than the evolution of objects through side effects, and they do not describe any strategies for higher-level, domain-specific means of representing and interacting with the space-time data.

The WHYLINE approach [14] describes an interaction model for debugging tools that allows programmers to directly answer “why” and “why not” questions about the space and time of an executed program. Similarly, our spacetime exploration model assists programmers in answering such questions. In comparison, we take a more open approach that fosters contextual information and encourages programmers not to express precise and close-ended questions but to select regions of the spacetime that they would like to explore in detail. Thus, both models involve different trade-offs between the gulf of execution and the gulf of interpretation of the results. In addition, the spacetime exploration model is not suited to answer questions about the dataflow of values.

8 Conclusion

We have proposed the idea of a universal tracing mode to make the information required for omniscient debugging available whenever needed in tools for exploratory programming. Furthermore, we proposed spacetime exploration, a novel interaction model for such tools that focuses on exploring the space and time of a program, effectively combining the concepts of object inspection and (process) debugging

into a single workflow. We have demonstrated its feasibility by implementing a prototypical spacetime inspector for the TRACEDEBUGGER in Squeak/Smalltalk and illustrating its use for two concrete scenarios. While in-situ omniscient debugging still comes with trade-offs for ensuring a responsive system with short feedback loops, we believe that programmers can get the feeling of having a powerful program tracer always available at their fingertips.

We believe that spacetime exploration can improve the experience of program exploration by keeping programmers focused and able to precisely express comprehension questions about space and time. Interactive domain-specific and task-specific views representing the objects under exploration can further provide rich contextual information, which connects to the well-known challenges and opportunities of software visualization. So far, we have optimized spacetime exploration for focusing on a single object. Programmers might also want to ask questions about systems that distribute relevant state across multiple objects or that do not provide a coherent state model, looking for refactoring opportunities. Further research is needed to extend our model to questions that depend on the dataflow of information such as when debugging infection chains.

In the future, we want to further generalize our interaction model and explore how it can be used as an overarching concept for all kinds of exploratory programming activities. To this end, we plan to make it possible to combine multiple views in the spacetime inspector, explore alternative means of filtering space and time, and extend our experience in connecting different domains to the spacetime inspector through domain-specific views. There have been several promising tool-construction frameworks (or concepts) that aim for flexible, task-specific view composition [4, 36]. Ultimately, we envision a perspective on symbolic (omniscient) debuggers that subsumes them under the spacetime exploration model by taking an object-centric view on the execution artifacts, and from where we can use our existing mechanics for filters and drill-downs through space and time to explore new means of navigation in omniscient debuggers.

Acknowledgments

We sincerely thank the anonymous reviewers for their detailed and valuable feedback. This work is supported by the HPI-MIT “Designing for Sustainability” research program⁹.

References

- [1] Mohammad R. Azadmanesh and Matthias Hauswirth. 2015. *Space-Time Views for Back-in-Time Debugging*. Technical Report 2015/02. University of Lugano. <http://sape.inf.usi.ch/sites/default/files/publication/usi-tr-2015-02.pdf>
- [2] Thomas Ball, Jung-Min Kim, Adam A. Porter, and Harvey P. Siy. 1997. If Your Version Control System Could Talk. In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, Vol. 11. 5 pages. https://www.researchgate.net/publication/2791666_If_Your_Version_Control_System_Could_Talk
- [3] Mary Beth Kery and Brad A. Myers. 2017. Exploring Exploratory Programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 25–29. <https://doi.org/10.1109/VLHCC.2017.8103446>
- [4] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Girba. 2015. The Moldable Inspector. In *2015 ACM 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. <https://doi.org/10.1145/2814228.2814234>
- [5] Andrei Chiş, Tudor Girba, and Oscar Nierstrasz. 2014. The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers. In *Software Language Engineering*, Benoit Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer International Publishing, Cham, 102–121. https://doi.org/10.1007/978-3-319-11245-9_6
- [6] Claudio Corrodi. 2016. Towards Efficient Object-Centric Debugging with Declarative Breakpoints. In *Post-proceedings of the 9th Seminar on Advanced Techniques and Tools for Software Evolution (CEUR Workshop Proceedings, Vol. 1791)*, Mircea Lungu, Anya Helene Bagge, and Haidar Osman (Eds.). CEUR-WS.org, Bergen, Norway, 32–39. <http://ceur-ws.org/Vol-1791/paper-04.pdf>
- [7] Jeffrey K. Czyz and Bharat Jayaraman. 2007. Declarative and Visual Debugging in Eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange* (Montreal, Quebec, Canada) (*eclipse '07*). Association for Computing Machinery, New York, NY, USA, 31–35. <https://doi.org/10.1145/1328279.1328286>
- [8] Hani Z. Girgis and Bharat Jayaraman. 2006. *JavaDD: A Declarative Debugger for Java*. Technical Report. Department of Computer Science and Engineering, University at Buffalo. <https://cse.buffalo.edu/tech-reports/2006-07.pdf>
- [9] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., USA. <https://dl.acm.org/doi/10.5555/273>
- [10] Simon F. Goldsmith, Robert O’Callahan, and Alex Aiken. 2005. Relational Queries over Program Traces. *ACM SIGPLAN Notices* 40, 10 (oct 2005), 385–402. <https://doi.org/10.1145/1103845.1094841>
- [11] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. 2006. Design and Implementation of a Backward-in-Time Debugger. In *NODE 2006 – GSEM 2006 (LNI, Vol. P-88)*, Robert Hirschfeld, Andreas Polze, and Ryszard Kowalczyk (Eds.). Gesellschaft für Informatik e.V., Bonn, 17–32. <https://dl.gi.de/items/ed80bc70-fcdd-4899-acc2-57f6de552aba>
- [12] Daniel Ingalls. 2020. The Evolution of Smalltalk: From Smalltalk-72 through Squeak. *Proc. ACM Program. Lang.* 4, HOPL, Article 85 (jun 2020), 101 pages. <https://doi.org/10.1145/3386335>
- [13] 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 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Atlanta, Georgia, USA). *SIGPLAN Not.* 32, 10, 318–326. <https://doi.org/10.1145/263700.263754>
- [14] Amy J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) (*ICSE '08*). Association for Computing Machinery, New York, NY, USA, 301–310. <https://doi.org/10.1145/1368088.1368130>
- [15] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. 1997. Query-Based Debugging of Object-Oriented Programs. *ACM SIGPLAN Notices* 32, 10 (1997), 304–317. <https://doi.org/10.1145/263698.263752>
- [16] Demian Lessa, Bharat Jayaraman, and Jeffrey K. Czyz. 2010. *Scalable Visualizations and Query-Based Debugging*. Technical Report. University at Buffalo, Department of Computer Science and Engineering.

⁹<https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html>

- <https://cse.buffalo.edu/tech-reports/2010-10.pdf>
- [17] Bil Lewis. 2003. Debugging Backwards in Time, In Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003). CoRR cs.SE/0310016, 11 pages. <https://doi.org/10.48550/ARXIV.CS/0310016>
- [18] S. Lewis. 1995. *The Art and Science of Smalltalk*. Prentice Hall. <https://rmod-files.lille.inria.fr/FreeBooks/Art/artAdded174186187Final.pdf>
- [19] Adrian Lienhard, Stéphane Ducasse, and Tudor Gîrba. 2009. Taking an Object-Centric View on Dynamic Information with Object Flow Analysis, In ESUG 2007 International Conference on Dynamic Languages (ESUG/ICDL 2007). *Computer Languages, Systems & Structures* 35, 1, 63–79. <https://doi.org/10.1016/j.cl.2008.05.006>
- [20] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. 2008. Practical Object-oriented Back-in-time Debugging. In *22nd European Conference on Object-oriented Programming (ECOOP 2008) (Lecture Notes in Computer Science, Vol. 5142)*. Springer Verlag, Paphos, Cyprus, 592–615. https://doi.org/10.1007/978-3-540-70592-5_25
- [21] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. 2004. A Versatile and User-Oriented Versioning File System. In *3rd USENIX Conference on File and Storage Technologies (FAST 04, Vol. 4)*. USENIX Association, San Francisco, CA, 115–128. https://www.usenix.org/legacy/publications/library/proceedings/fast04/tech/full_papers/muniswamy/muniswamy.pdf
- [22] Donald A. Norman. 1986. *Cognitive Engineering*. Lawrence Erlbaum Associates, Hillsdale, NJ, USA, 31–61. https://www.researchgate.net/profile/Donald-Norman-3/publication/235616560_Cognitive_Engineering/links/0c960536c18209b825000000/Cognitive_Engineering.pdf
- [23] Michael Perscheid and Robert Hirschfeld. 2014. Follow the Path: Debugging Tools for Test-Driven Fault Navigation. In *2014 Tool Demo Track of the 1st Conference on Software Maintenance, Reengineering, and Reverse Engineering Software Evolution Week (CSMR-WCRE '14)*. IEEE Computer Society, Los Alamitos, CA, USA, 446–449. <https://doi.org/10.1109/CSMR-WCRE.2014.6747215>
- [24] Michael Perscheid, Bastian Steinert, Robert Hirschfeld, Felix Geller, and Michael Haupt. 2010. Immediacy through Interactivity: Online Analysis of Run-Time Behavior. In *2010 17th Working Conference on Reverse Engineering*. 77–86. <https://doi.org/10.1109/WCRE.2010.17>
- [25] Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2013. Expositor: Scriptable Time-Travel Debugging with First-Class Traces. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, San Francisco, CA, USA, 352–361. <https://doi.org/10.1109/ICSE.2013.6606581>
- [26] Alex Potanin, James Noble, and Robert Biddle. 2004. Snapshot Query-Based Debugging. In *2004 Australian Software Engineering Conference. Proceedings*. IEEE Computer Society, Melbourne, Australia, 251–259. <https://doi.org/10.1109/ASWEC.2004.1290478>
- [27] Guillaume Pothier and Éric Tanter. 2009. Back to the Future: Omniscient Debugging. *IEEE Software* 26, 6 (2009), 78–85. <https://doi.org/10.1109/MS.2009.169>
- [28] Patrick Rein and Robert Hirschfeld. 2018. The Exploration Workspace: Interleaving the Implementation and Use of Plain Objects in Smalltalk. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming (Nice, France) (Programming '18)*. Association for Computing Machinery, New York, NY, USA, 113–116. <https://doi.org/10.1145/3191697.3214339>
- [29] 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 (07 2019), 33 pages. <https://doi.org/10.22152/programming-journal.org/2019/3/1>
- [30] Jakob Reschke, Marcel Taeumel, Tobias Pape, Fabio Niephaus, and Robert Hirschfeld. 2018. *Towards Version Control in Object-Based Systems*. Vol. 121. Universitätsverlag Potsdam, Potsdam, Germany. https://hpi.de/fileadmin/user_upload/hpi/dokumente/publikationen/technische_berichte/tbhpi121.pdf
- [31] Jorge Ressaia, Alexandre Bergel, and Oscar Nierstrasz. 2012. Object-centric Debugging. In *2012 34th International Conference on Software Engineering (ICSE)*, Martin Glinz, Gail Murphy, and Mauro Pezze (Eds.). IEEE, IEEE Computer Society, Washington, DC, USA, 485–495. <https://doi.org/10.1109/ICSE.2012.6227167>
- [32] Tim Rowledge. 2001. A Tour of the Squeak Object Engine. In *Squeak: Open Personal Computing and Multimedia* (1st ed.), Mark J. Guzdial and Kimberly M. Rose (Eds.). Prentice Hall PTR, USA. <https://rmod-files.lille.inria.fr/FreeBooks/CollectiveNBlueBook/Rowledge-Final.pdf>
- [33] David W. Sandberg. 1988. Smalltalk and Exploratory Programming. *SIGPLAN Not.* 23, 10 (1988), 85–92. <https://doi.org/10.1145/51607.51614>
- [34] Rodrigo Schulz, Fabian Beck, Jhonny Wilder Cerezo Felipez, and Alexandre Bergel. 2016. Visually Exploring Object Mutation. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)* (Raleigh, NC, USA). IEEE, 21–25. <https://doi.org/10.1109/VISSOFT.2016.21>
- [35] Ben Shneiderman and Catherine Plaisant. 2005. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (4th ed.). Pearson Education, India. <http://seu1.org/files/level5/IT201/Book%20-%20Ben%20Shneiderman-Designing%20the%20User%20Interface-4th%20Edition.pdf>
- [36] Marcel Taeumel. 2020. *Data-Driven Tool Construction in Exploratory Programming Environments*. Ph. D. Dissertation. University of Potsdam, Digital Engineering Faculty, Hasso Plattner Institute. <https://doi.org/10.25932/publishup-44428>
- [37] 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 International Publishing, Cham, 111–145. https://doi.org/10.1007/978-3-031-09297-8_7
- [38] Christoph Thiede and Patrick Rein. 2023. *Squeak by Example*. Vol. 6.0. Lulu. <https://www.lulu.com/shop/patrick-rein-and-christoph-thiede/squeak-by-example-60/paperback/product-8vr2j2.html> ISBN 978-1-4476-2948-1.
- [39] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. Object-Centric Time-Travel Debugging: Exploring Traces of Objects. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming (Tokyo, Japan) (<Programming> '23)*. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3594671.3594678>
- [40] Arian Treffer and Matthias Uflacker. 2016. The Slice Navigator: Focused Debugging with Interactive Dynamic Slicing. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE Computer Society, Los Alamitos, CA, USA, 175–180. <https://doi.org/10.1109/ISSREW.2016.17>
- [41] David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the Experience of Immediacy. *Commun. ACM* 40, 4 (apr 1997), 38–43. <https://doi.org/10.1145/248448.248457>
- [42] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. 2020. A Principled Approach to REPL Interpreters. In *Proceedings of the 2020 ACM 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, 84–100. <https://doi.org/10.1145/3426428.3426917>

Received 2023-04-28; accepted 2023-08-11