Bringing Objects to Life: Supporting Program Comprehension through Animated 2.5D Object Maps from Program Traces

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

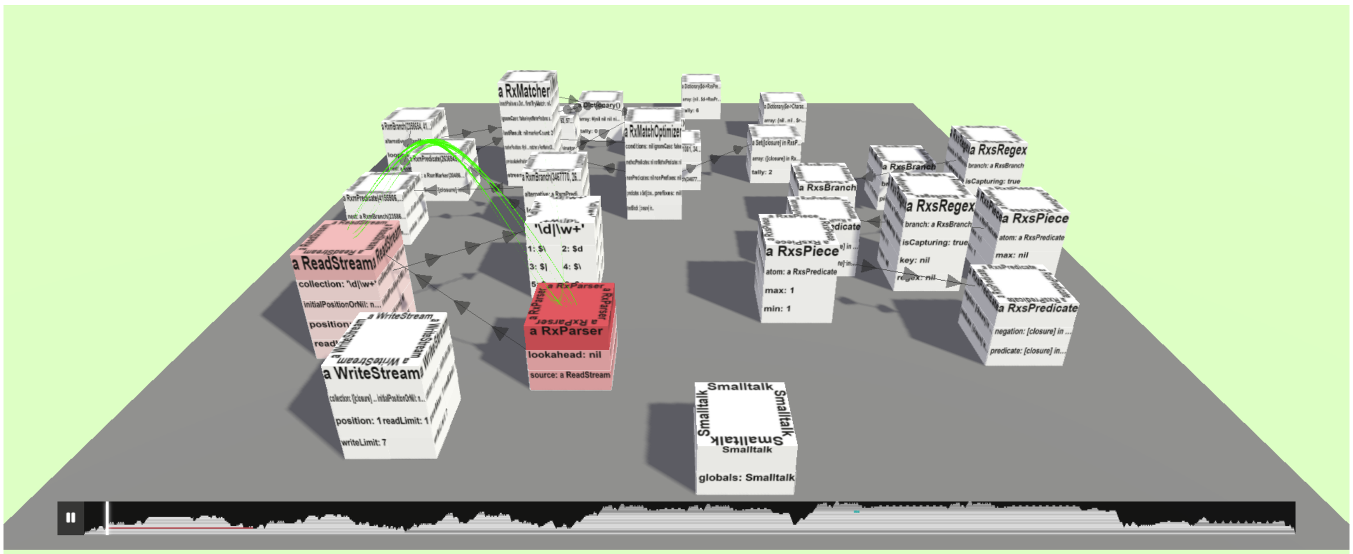

Figure 1: TODO

ABSTRACT

Program comprehension is a key activity in software development. Several visualization approaches such as software maps have been proposed to support programmers in exploring the architecture of software systems, while little attention has been paid to the exploration of program behavior and programmers still rely on traditional code browsing and debugging tools. We propose a novel approach for visualizing program behavior through *animated 2.5D object maps* that depict particular objects and their interactions from a program trace. We describe our implementation of this approach and evaluate it for different program traces through an experience report and performance measurements. Our results indicate that our approach can be beneficial for program comprehension tasks, but that further research is needed to improve scalability and usability.

CCS CONCEPTS

• **Human-centered computing** → **Visualization techniques**; • **Software and its engineering** → Software maintenance tools.

KEYWORDS

software visualization, software maps, object-oriented programming, program comprehension, omniscient debugging

1 INTRODUCTION

Exploring and understanding software systems play a central role in software development. Programmers frequently get thrown into unknown systems that they want to fix, change, or extend. For this, they need to build up a mental model that connects the system's visible behavior to its high-level architecture and low-level implementation artifacts.

Traditionally, programmers explore software systems by reading their source code. An alternative approach is to explore the system's behavior by example: programmers can start by invoking the system with a particular input or by running a test case and then use a debugger to step through the program's execution, identify relevant units and actors, and explore their interactions. As traditional debuggers are constrained to the temporal execution order of the program, *omniscient debuggers* (also referred to as *time-travel debuggers* or *back-in-time debuggers*) exist that record a *program trace* and allow programmers to explore the program's behavior in a non-linear fashion [37, 27, 50, 41, 62]. However, omniscient debuggers are not well suited for exploring large program traces involving several subsystems and dozens of interacting objects: while their fine-grained display of source code and variables is useful for debugging-related activities, it impedes the exploration of the system's high-level architecture and behavior.

On the other hand, several visualization approaches have been proposed to support programmers in exploring the architecture of software systems. In particular, *software maps* that display the static structure of systems using several metaphors such as cities or forests were found to be useful for program comprehension tasks [68, 2, 43]. Yet, most approaches neglect the dynamic behavior of systems and take a coarse-grained view of their structure, which makes them unsuitable for developing a mental model of the system's behavior that situates particular interacting objects and connects them to the overall functioning of the system.

To bridge this gap between coarse-grained static software maps and fine-grained omniscient debugging views, we propose a novel approach for visualizing the behavior of object-oriented software systems through animated 2.5D maps depicting particular objects and their interactions from a program trace. In particular, we make the following contributions:

(1) We present a novel visualization approach for object-oriented program behavior through animated 2.5D object maps.
(2) We describe the implementation of our prototype TRACE-4D that applies this approach using program traces from a Squeak/Smalltalk environment and the THREE.js 3D library.
(3) We discuss the potential and limitations of our approach by reporting on our experience with it and by evaluating the performance of our implementation for different program traces.

We make all artifacts of this work available at a public repository[1].

The remainder of this paper is structured as follows: in section 2, we discuss related work on software maps, object-oriented programs, and program traces. In section 3, we present our visualization approach for program traces. In section 4, we describe our implementation of this approach. In section 5, we describe the use of our visualization tool by an example. In section 6, we discuss the potential and limitations of animated object maps through an experience report and a performance evaluation. Finally, we conclude and discuss future work in section 7.

2 RELATED WORK

Several approaches for visualizing the architecture and behavior of software systems have been proposed in the past. In the broad field of program visualization [47, 58, 60], *algorithm animation* is an early approach that mainly focuses on visualizing procedural algorithms and data structures in educational contexts [8]. During the last decades, more approaches have been proposed that allow to create general-purpose visualizations for the architecture and behavior of arbitrary software systems [53, 11, 12, 17].

2.1 Software Architecture Visualization

The term *software maps* describes a family of approaches that use metaphors from cartography to visualize the architecture of software systems.

Treemaps. Treemaps display the static structure of software systems by visualizing their hierarchical organization of packages and classes, folders and files, autc. as a nested set of shapes [42, 43]. They offer different visual variables such as the size, color, and position of the shapes to encode additional information about the system's size or evolution. Shapes are usually rectangles but can also be polygons as in Voronoi tesselation treemaps [6]. A popular modern type of treemaps is *2.5D treemaps* that add a third dimension to the visualization by transforming each shape into a right prisma (usually cuboid) of a variable height. Many approaches use the *software city* metaphor to style the cuboids of a 2.5D treemap as buildings of a city [18, 68, 1, 45, 28, 43].

Topic maps. Unlike treemaps, *topic maps* do not display the programmer-specified organization of a software system but use natural language processing techniques such as source code topic models, latent dirichlet allocation, and multidimensional scaling to arrange units of the software in a 2D or 3D graph [4]. Different metaphors have been proposed to embody these graphs in a map, including boardgames [3] and landscapes such as forests [2] and galaxies [5].

Animated software maps. Next to using static visual variables, some approaches enrich software maps with animations to display dynamic information over time [36, sec. 3.4]. Dynamic information can refer to the behavior or evolution of software: for instance, EvoSpaces [18] highlights classes in a software city when they are activated, while DynaCity [15], ExplorViz [32], SynchroVis [67], and others [13] also draw connections between modules to visualize dataflow between them; [35] gradually constructs a software city and updates the geometries and colors of buildings to represent development activity, and Gource [10] enhances the construction animation of a file tree with moving avatars representing code authors. Some approaches allow programmers to monitor a system in real-time [20] while others replay a previously recorded trace of software activity [18].

2.2 Entity-Centric Behavior Visualization

To provide visual insights into the behavior of software, a natural choice is to attribute behavior to different entities of the software. Entities can be organizational units such as modules or classes but also individual object instances of object-oriented programs.

Object graphs. Several tools allow programmers to explore relevant portions of a program's object graph [44, 23]. Some graphs resemble the look of UML object diagrams and provide details about objects's internal state while others choose more compact representations. To reduce the visual complexity of graph displays, some tools provide programmers with means for filtering objects based on their organization or relation to program slices [34, 26].

Communication flow. Call graphs and *control-flow graphs* are two popular ways for displaying entities with their mutual dynamic interactions or communications [16, 34, 36, 52, 64, 7, 51]. Entities can be nodes from an object graph or organizational units such as classes or modules. Avid and PathObjects [56] provide animated object graphs where users can explore the control flow interactively. [7] merges the stack frames from a control-flow graph and the nodes from an object graph into a single *memeograph* that can be explored through animation.

In contrast to traditional call graphs, some works have proposed peripheral, hierarchical layouts of nodes such as Extravis' *circular*

[1] https://github.com/LinqLover/trace4d

bundle views [14] or [46]'s 3D hyperbolic layout that provide better scaling for highly connected graphs. Another representation of inter-entity communication is to provide full adjacency matrices of the call graph [49].

Dataflow. Another perspective that can be taken on the object graph is how state is transferred through the system. The Whyline approach allows programmers to ask questions about why certain behaviors did or did not happen or where certain values came from and presents the answers in a sliced control-flow graph [31]. [38] proposes an *inter-unit flow view* that displays the amount of data or objects exchanged between different classes or modules in a directed weighted graph; this graph can also be embedded into a traditional call graph [39].

State changes. [39] also proposes a *side-effects graph* [19] (also referred to as *test blueprints* [40]) that displays connections between objects changing each other's state. Similarly, *object traces* describe a way to slice a call graph for exploring the state evolution of single objects [62, 63].

2.3 Time-Centric Behavior Visualization

Next to the communication between entities, another perspective that visualizations commonly take on software behavior is the temporal order of program execution.

Call trees. A call tree is a hierarchy of stack frames or message sends that can be gained from a program trace. Besides naive graph representations of this data structure, several approaches display call trees using hierarchical layouts such as treemaps, sunbursts, or *icicle plots* [33, 65, 69]. Similarly to icicle plots, *flame graphs* show the historical call stack over time, but they also assign colors to stack frames for displaying additional performance data from profiling tools [24].

Sequential displays. UML sequence diagrams are a traditional approach for displaying communication between objects over time. Several tools adopt [59] and extend [26] this diagram type: for instance, ISVis' *information mural* [30] and Extravis' *massive sequence view* [14] derive miniaturized versions of a sequence diagram [36, sec. 3.4], and Ovation [16] detects execution patterns to reduce sequence diagrams [26].

3 VISUALIZATION APPROACH

In this section, we describe the prerequisites and the design of our proposed visualization approach.

3.1 Data Model

The data source of our visualization is the program trace of an object-oriented program. In this programming paradigm, all behavior is described as *messages* sent from one object to another. Each object is characterized by its *identity* which distinguishes it from all other objects in the system, its *state* which is represented by its fields such as array elements and instance variables, and its *behavior* which is implemented by methods that are invoked to receive messages [63].

We assume a minimal data model of the program trace (fig. 2): the *call tree* is represented as a composite structure of *stack frames*

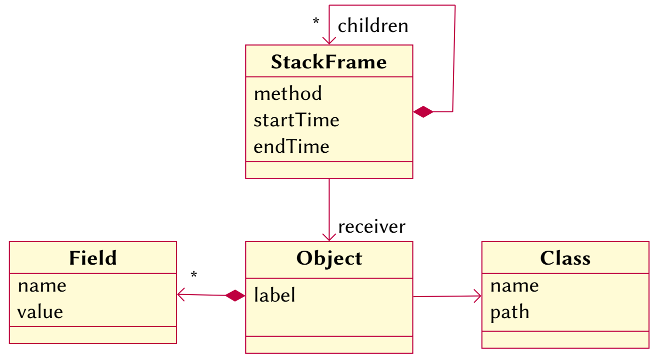

Figure 2: TODO

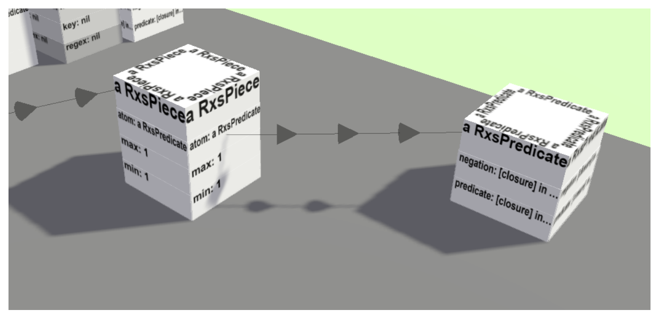

Figure 3: TODO

each of which specifies a time interval, an invoked method, and a receiver object. Each *object* is assigned a label, a list of named fields, and a class. Each *class* is described through a name and an organizational path in the file or package structure of the software system. We neglect runtime changes to the state, label, or class membership of objects as well as metaprogramming specifics such as the implementation of classes or methods as objects.

3.2 Visual Mapping

We describe the design of our visualization and the mapping of parts from the program trace to elements and visual variables of our visualization (fig. 1). At the highest level, an animated 2.5 object map is an interactive information landscape that displays objects and their interactions from the program trace. Users can replay the program trace and watch the activation and interaction of objects. They can unrestrictedly navigate through the visual scene using their keyboard and pointing devices and view the map from all sides.

Objects. Each object is represented as a square cuboid *block* entity that displays the label and fields of the object (fig. 3). To maximize legibility from any perspective, the label is repeated on all four sides and in four orientations on the top of the block. Fields are displayed as *plates* that are arranged in a row-wise uniform-sized grid layout and repeated on each side of the block for better legibility. References between objects are rendered as *directed arrows* from the closest plate of the referencing field to the closest label of the referenced object's entity. To indicate the direction of arrows,

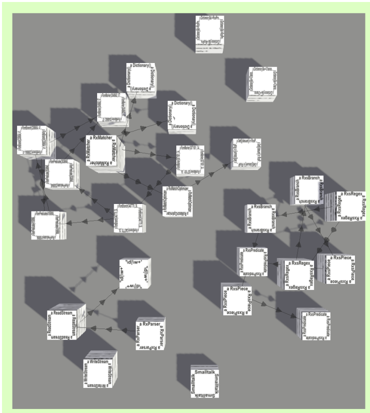

Figure 4: TODO

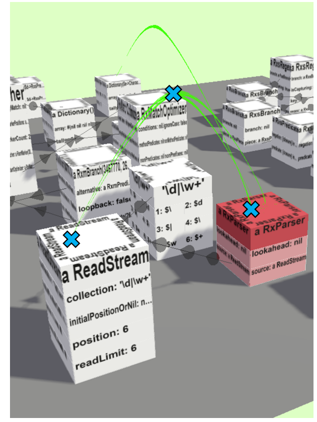

Figure 5: TODO

Table 1: TODO

w_{class}	w_{org}	w_{ref}	w_{comm}	w_{repulse}	w_{center}
0.001	$F \mapsto 0.005\left(\log_{10}(F)+1\right)$	0.1	0.00001	0.2	0.00142

we place between one and ten evenly distributed *chevrons* on the arrow line; each chevron is displayed as a cone whose direction can be recognized from any perspective.

Object graph. To arrange object blocks in the 2.5D object map, we define a force-directed graph layout [21]. Between each pair of object blocks (a, b), we apply several *weighted attractive forces* based on the class membership, the organizational proximity of classes, and the references and communication between objects (fig. 4):

$$\begin{aligned}
F_{\text{class}}(a,b) &= w_{\text{class}}\left(\begin{cases}1, & \text{if class}(a) = \text{class}(b);\\ 0, & \text{otherwise.}\end{cases}\right),\\
F_{\text{org}}(a,b) &= w_{\text{org}}\left(\text{LCP}^{2}\left(\text{org}^{3}(a), \text{org}(b)\right)\right),\\
F_{\text{ref}}(a,b) &= w_{\text{ref}}\left(\left|\left\{(k,v) \in \text{fields}(a) \;\middle|\; v = b\right\}\right|\right),\\
F_{\text{comm}}(a,b) &= w_{\text{comm}}\left(\left|\left\{\text{frame } f \;\middle|\right.\right.\right.\\
&\qquad \left.\left.\left. f.\text{receiver} = a \wedge f.\text{parent.receiver} = b\right\}\right|\right).
\end{aligned} \tag{1}$$

In addition to the attractive forces, we define globally weighted *repulsion* and *centripetation* forces on all blocks to control the entropy of the graph, and we define *radial constraints* to avoid collisions between blocks.

We provide an empirical base configuration for all force weights but enable users to override them for specific program traces. By default, we prioritize reference forces the highest and organizational forces the lowest with a distance of six orders of magnitudes and scale organizational forces logarithmically (table 1). This configuration fosters a state-centric layout of the object graph while leaving a margin for the characteristic of particular program traces (e.g., their ratio between intrinsic and extrinsic state [22, p. 218ff]) towards a more dataflow-driven layout. Additionally, users can drag and drop blocks to override the layout. To reduce response times and maintain an experience of immediacy [57, chap. 11, 66], we render the graph continuously before the force simulation has converged.

Object selection. Usually, even after restricting the object graph to the receivers from the call tree (section 3.1), only a small part of it is relevant for comprehending the high-level behavior of a program while many other objects fulfill lower-level implementation details. In our visualization, we use a filtering system for excluding objects based on their label, class, or organization. Similarly to the layout configuration (object graph), we provide an empirical default configuration that excludes certain base objects such as collections, booleans, and numbers, but allow users to customize these filters.

Object behavior. The color of each object block displays its recent activity: *inactive* blocks are colored in a neutral light gray while *active* blocks whose objects have received a message recently are highlighted in a bright red (fig. 5). After the control flow passes on to other objects, blocks linearly fade back to the base color within one second, thus applying a single-hue continuous sequential color scheme by Brewer et al.[3]

Next to the color coding, a *trail* connects the most recent $k = 15$ object activations to support the delayed observation of short activations and the recognition of exact activation order. The trail curve is based on a centripetal Catmull-Rom spline [9] whose control points are placed on the top of each relevant block and are alternated with intermediate points between blocks. Block control points are normally randomized to make multiple activations of the same object distinguishable. Intermediate control points are vertically elevated to give the curve a wave-like shape that makes activated objects identifiable. The direction of the trail is displayed by continuously moving it to the next object and applying a linear translucency gradient to fade out the tail of the curve.

Timeline. The object map integrates a *timeline* overlay at the bottom of the viewport that provides a time-centric navigational aid. The timeline consists of two widgets stacked on top of each other (fig. 6): a *player* with a slider and a play/pause button indicates

[2]LCP(u, v): Largest common prefix of two sequences u and v.
[2]org(o): Organizational path to an object o's class (e.g., a file path).

[3]Cynthia Brewer and Mark Harrower. 2013 – 2021. ColorBrewer: Color Advice for Cartography. Pennsylvania State University. URL: https://colorbrewer2.org/

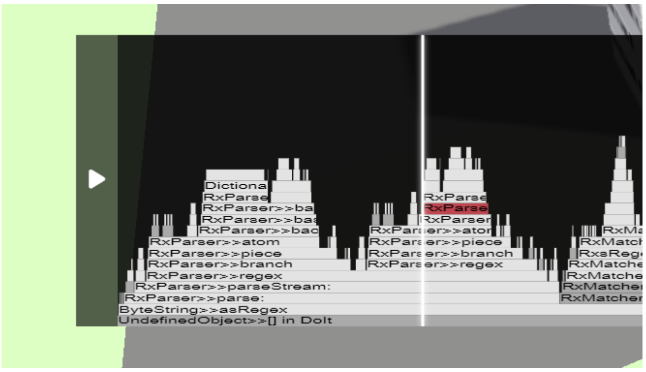

Figure 6: TODO

the current point in time of the program trace and allows to control the time and animation state. Behind the player, a collapsed *flame graph* displays the course of the call stack depth. Users can resize the timeline to explore the full call tree hierarchy and examine single frames in the flame graph.

Both the flame graph and the object map are interactively connected, i.e., users can hover an object in the map to discover all of its activations in the timeline, or vice versa, they can click on a frame to move the trail in the map to the relevant activation of the object. Thus, object map and timeline provide two orthogonal means for navigating through the object-oriented program trace with different granularities.

4 IMPLEMENTATION

Here we describe the implementation of animated 2.5D object maps in our prototype TRACE4D that displays program traces from a Squeak/Smalltalk environment in a web application.

Program tracing. Squeak/Smalltalk is an interactive programming environment that is based on the object-oriented paradigm (everything is an object, including classes, methods, and stack frames) and offers programmers rich control for inspecting and manipulating all parts of the system (by instrumenting method objects, recording stack frame objects, etc.) [29, 55, 61]. We use the TraceDebugger[4] [62], which is an omniscient debugger for Squeak, to record a program trace of interesting behavior such as compiling a method, matching a string against a regular expression, or handling user events in a graphical user interface (GUI).

We serialize the resulting program trace consisting of a call tree, an object graph, and a class hierarchy and export it to a JSON file. To retrieve the fields for each object, we use Squeak's built-in inspector tool [61, chap. 6, sec. 3] which collects all instance variables or array elements from each object but also provides higher-level views on the state of known domain objects; for instance, a dictionary will not be presented with its internal overallocation array structure but with a more comprehensible collection of key-value pairs. Regarding the objects referenced as values from fields, we only include those objects in the serialization that receive at least one message in the program trace but only store a flat string representation of any other objects to avoid traversing the entire object graph of the system whose largest part is not relevant to the particular program trace.

Visualization. We implement the visualization frontend of TRACE4D as a JavaScript web application. The web app retrieves a serialized program trace and offers a programmatical interface for customizing the visual configuration (section 3.2). To build the 2.5D object map, we generate and display a 3D scene from the program trace using the JavaScript 3D library THREE.JS[5] and layout the object blocks using the `d3-force` module of the visualization framework D3.JS[6]. To build the timeline, we create a flame graph using the `d3-flame-graph` plugin for D3.JS[7] and combine it with a custom HTML widget for the player controls[8]. To animate the visualization, we traverse the call tree with a configurable speed (defaulting to 50 bytecode instructions per second) and update the color highlights and trail for activated objects at each animation tick.

5 USE CASE: EXPLORING INTERNALS OF A REGULAR EXPRESSION ENGINE

In this section, we illustrate how a programmer can use the TRACE4D visualization to explore the way a regular expression engine constructs a matcher from a pattern. The `Regex` package in Squeak provides a Smalltalk-specific flavor of regular expressions. To construct a matcher, the package first parses the pattern string into an abstract syntax tree (AST) and then compiles the AST into a non-deterministic finite automaton (NFA). In this example, our programmer visualizes the construction of the simple regular expression `\d|\w+` to gain a closer understanding of the involved subsystems and their interactions.

To create the visualization, the programmer records and exports a trace of the program `'\d|\w+' asRegex` in Squeak and loads it into the TRACE4D web app[9]. As the visualization loads, she can see about 25 objects moving around in the object map and arranging themselves into a semi-structured graph within a few seconds (fig. 1). By navigating through the scene, she discovers several meaningful objects and clusters of objects:

- the pattern string `'\d|\w+'`;
- an `RxParser` object accessing the string through a `ReadStream`;
- eight objects referencing each other whose class names start with the prefix `Rxs`, identifying them as nodes of the AST;
- a `RxMatcher` object surrounded by six objects whose class names start with `Rxm`, identifying them as states of the matcher's NFA;
- several other loosely structured objects, including an `RxMatchOptimizer` object, four `Dictionary`s, and a `Set`.

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

[5] https://threejs.org/

[6] https://d3js.org/

[7] https://github.com/spiermar/d3-flame-graph

[8] As `d3-flame-graph` at the time of writing does not support a notion of starting points but only lengths for frames, we inject auxiliary transparent frames into the flame graph to adjust the horizontal layout of actual frames (see https://github.com/spiermar/d3-flame-graph/issues/227).

[9] The interactive visualization of the described program trace is available at https://linqlover.github.io/trace4d/?trace=traces/regexParse.json and in the Wayback Machine of the Internet Archive.

Table 2: Ratings of our experience with animated object maps for program comprehension (??). We gained the most insights from smaller program traces that thoroughly model behavior through communication between objects and avoid many similar objects.

Program	Configuration effort	Clarity of objects	Object layout	Animation	Program comprehension
`Regex` *engine*					
• Construction	+	+	+	+	+
• Matching	+	+	+	○	+
`Morphic` *UI framework*					
• Event handling	−	−	○	○	○
• Layouting	○	○	+	○	−
`Inspector` tool initialization	−	−	−	−	−
HTML parsing	○	+	+	+	+

After she has gained a rough overview of the object graph, she starts the animation of the program trace through the player in the timeline. By observing the trail of object activations and the position of the cursor in the timeline (default running time: 77 seconds), she notices the following rough segments of the program execution:

(1) Invoked by the pattern string, the parser dominates the first third of the program, accesses the pattern through the `ReadStream`, and talks to the AST nodes, presumably to initialize them.
(2) Next, the matcher gets active and accesses the AST nodes and the NFA states simultaneously, presumably to compile the AST into the NFA.
(3) For the remaining half of the program time, the match optimizer is active, accessing the AST again and talking to the set.

Thus, our programmer was able to gain an initial overview of the different parts of the `Regex` package and their collaboration to realize the construction of the matcher. Besides, she also noticed that almost 50% of the time were spent in the match optimizer. Without a closer idea of the role of this object, she suspects this step to be a bottleneck of the construction and wonders whether the optimization might be optional and could be skipped for certain uses of the regular expression. To dive deeper into the implementation of the `Regex` package, she expands the flame graph of the timeline, identifies a few entry point methods of the objects that she found most interesting (e.g., `RxParser»parseStream:` or `RxMatchOptimizer»initialize:ignoreCase:`), and opens them in the Squeak environment to browse their source code.

6 DISCUSSION

In this section, we discussion the potential and limitations of our visualization approach by reporting on our experience and evaluating the performance of the TRACE4D prototype for six different use cases.

6.1 Experience Report

To estimate the use of animated object maps for program comprehension, we explored six different program traces from the domains of string processing, GUIs, and programming tools in the TRACE4D prototype and gave a reasoned rating of our experience with each example on a three-point Likert scale for five different criteria regarding the usability, clarity, and insightfulness of the visualization (table 2). We provide a full protocol of the experience report in ??.

Suitable traces. We made better experiences when using the visualization for smaller program traces such as different string processing examples. On the contrary, we were more challenged trying to understand the behavior of larger program traces such as operations in a GUI system or in a programming tool. In general, we found animated object maps most practical for systems that thoroughly adhere to the principles of object-oriented design by defining many fine-grained, highly coherent objects and describing behavior through extensive communication between these objects. On the other hand, program traces involving many homogenous objects or unrelated subsystems contain more repetitive or irrelevant elements and are typically less suited for exploration through animated object maps. Thus, it is the task of programmers to condense interesting behavior to a minimum program by reducing inputs and eliminating dependencies, as they already use to do when preparing a minimal reproducible example to locate the defect in a program.

Program comprehension. For suitable program traces, we were able to gain several kinds of insights and benefits from the visualization: we managed to discover characteristic regions of the object graph (e.g., the input, the AST, and the NFA for the regular expression use case, section 5) as well as significant segments of the program behavior (e.g., the parsing, compiling, and optimization stages in the regular expression use case). Based on this overview, we could develop or refine our mental model of the explored system's functioning and connect it to particular classes and objects in their implementation. Further, the interactive visualization helped us explore and analyze communication patterns, reflect upon the system design, and share and discuss these mental models with other developers.

Object graph layout. The structure of the object graph layout is determinant for the comprehension of the program state. Our force-directed graph approach provides a simple yet effective way to describe a layout based on different static and behavioral relations between objects and allows different kinds of relations to dominate the layout depending on the characteristics of the program trace. Especially for smaller program traces, the resulting layout allowed us to distinguish between essential regions of the object graph. Still, the overall structure of the force-directed layout could be considered too weak for an optimal visual intuition.

As an alternative to the force-directed layout, we consider clustering objects into discrete groups that could be displayed in a clearer structure through color coding or a hierarchical layout of objects. Clusters could either be collected from the existing force-directed layout or based on other distance metrics for objects such as their class organization, their communication patterns, or embedding representations derived from their source code or documentation.

Limitations. A general challenge of information visualization lies in reducing the complexity of the underlying data to a comprehensible but meaningful level [54]. For animated object maps, this challenge manifests as users being overwhelmed by the amount of objects and messages in the visualization of larger program traces. To address this challenge, our approach already provides a configuration interface that allows users to reduce the complexity of the visualization by filtering objects or improving the structure of the object graph. Still, the configuration requires manual effort and is thus a barrier for users to overcome. To lower this barrier, we aim to improve the convenience of the configuration interface in our prototype by allowing users to refine the configuration directly in the running visualization; however, we see more potential in further research on automatic configuration techniques that can generate a suitable configuration for individual program traces.

Another source of complexity in animated object maps is the cluttered communication between different objects, e.g., lengthy handshakes between objects or messages that are not relevant for the high-level program behavior. To address this challenge, we want to apply trace summarization techniques to eliminate implementation details from the underlying program trace [25, 48].

6.2 Evaluation of Performance

Another challenge regards the technical performance of the visualization which affected our experience for larger program trace. While computational efficiency was not a design goal for our current implementation of the TRACE4D prototype, it already delivers practical performance for most of our considered program traces; still, there is a need for optimizing the frame rate, graphic memory consumption, and saving/loading times of program traces (table 3).

To speed up the saving/loading times, we see great optimization potential in applying object filters in the backend (IDE) prior the serializing the program trace. To improve the responsiveness of the visualization, we consider replacing the current force-simulation libary `d3-force` by a more efficient alternative and extracting it from the UI thread into a parallel web worker. Finally, we believe that the 3D rendering performance could be improved significantly through several optimizations such as applying a level-of-detail strategy, optimizing the memory management of the application, or reducing the visual complexity of the scene.

7 CONCLUSION

REFERENCES

[1] Susanna Ardigò, Csaba Nagy, Roberto Minelli, and Michele Lanza. 2021. Visualizing data in software cities. In *2021 Working Conference on Software Visualization (VISSOFT)*, 145–149. DOI: 10.1109/VISSOFT52517.2021.00028.

[2] Daniel Atzberger, Tim Cech, Merlin Haye, Maximilian Söchting, Willy Scheibel, Daniel Limberger, and Jürgen Döllner. 2021. Software forest: a visualization of semantic similarities in source code using a tree metaphor. In (Feb. 2021), 112–122. DOI: 10.5220/0010267601120122.

[3] Daniel Atzberger, Tim Cech, Adrian Jobst, Willy Scheibel, Daniel Limberger, Matthias Trapp, and Jürgen Döllner. 2022. Visualization of knowledge distribution across development teams using 2.5 d semantic software maps. In *VISIGRAPP (3: IVAPP)*, 210–217.

[4] Daniel Atzberger, Tim Cech, Willy Scheibel, Daniel Limberger, and Jürgen Döllner. 2023. Visualization of source code similarity using 2.5 d semantic software maps. In *Computer Vision, Imaging and Computer Graphics Theory and Applications: 16th International Joint Conference, VISIGRAPP 2021, Virtual Event, February 8–10, 2021, Revised Selected Papers*. Springer, 162–182.

[5] Daniel Atzberger, Willy Scheibel, Daniel Limberger, and Jürgen Döllner. 2021. Software galaxies: displaying coding activitiesusing a galaxy metaphor. In *Proceedings of the 14th International Symposium on Visual Information Communication and Interaction* (VINCI '21) Article 18. Association for Computing Machinery, Potsdam, Germany, 2 pages. ISBN: 9781450386470. DOI: 10.1145/3481549.3481573.

[6] Michael Balzer, Oliver Deussen, and Claus Lewerentz. 2005. Voronoi treemaps for the visualization of software metrics. In *Proceedings of the 2005 ACM Symposium on Software Visualization* (SoftVis '05). Association for Computing Machinery, St. Louis, Missouri, 165–172. ISBN: 1595930736. DOI: 10.1145/1056018.1056041.

[7] Peter Boothe and Sandro Badame. 2011. Animation of object-oriented program execution. In *Proceedings of Bridges 2011: Mathematics, Music, Art, Architecture, Culture*. Reza Sarhangi and Carlo H. Séquin, (Eds.) Tessellations Publishing, Phoenix, Arizona, 585–588. ISBN: 978-0-9846042-6-5. http://archive.bridgesmathart.org/2011/bridges2011-585.html.

[8] Marc H Brown and Robert Sedgewick. 1984. A system for algorithm animation. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 177–186.

[9] Edwin Catmull and Raphael Rom. 1974. A class of local interpolating splines. In *Computer Aided Geometric Design*. Robert E. Barnhill and Richard F. Riesenfeld, (Eds.) Academic Press, 317–326. ISBN: 978-0-12-079050-0. DOI: https://doi.org/10.1016/B978-0-12-079050-0.50020-5.

[10] Andrew H. Caudwell. 2010. Gource: visualizing software version control history. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (OOPSLA '10). Association for Computing Machinery, Reno/Tahoe, Nevada, USA, 73–74. ISBN: 9781450302401. DOI: 10.1145/1869542.1869554.

[11] Yung-Pin Cheng, Jih-Feng Chen, Ming-Chieh Chiu, Nien-Wei Lai, and Chien-Chih Tseng. 2008. Xdiva: a debugging visualization system with composable visualization metaphors. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (OOPSLA Companion '08). Association for Computing Machinery, Nashville, TN, USA, 807–810. ISBN: 9781605582207. DOI: 10.1145/1449814.1449869.

[12] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. 2014. The moldable debugger: a framework for developing domain-specific debuggers. In *Software Language Engineering*. Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, (Eds.) Springer International Publishing, Cham, 102–121. ISBN: 978-3-319-11245-. DOI: 10.1007/978-3-319-11245-9_6.

[13] Marcus Ciolkowski, Simon Faber, and Sebastian von Mammen. 2017. 3-d visualization of dynamic runtime structures. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement* (IWSM Mensura '17). Association for Computing Machinery, Gothenburg, Sweden, 189–198. ISBN: 9781450348539. DOI: 10.1145/3143434.3143435.

[14] Bas Cornelissen, Andy Zaidman, Arie van Deursen, and Bart van Rompaey. 2009. Trace visualization for program comprehension: a controlled experiment. In *2009 IEEE 17th International Conference on Program Comprehension*, 100–109. DOI: 10.1109/ICPC.2009.5090033.

[15] Veronika Dashuber and Michael Philippsen. 2022. Trace visualization within the software city metaphor: controlled experiments on program comprehension. *Information and Software Technology*, 150, 106989. DOI: https://doi.org/10.1016/j.infsof.2022.106989.

[16] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. 1998. Execution patterns in object-oriented visualization. In *Proceedings of the 4th Conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4* (COOTS'98). USENIX Association, Santa Fe, New Mexico, 16. DOI: 10.5555/1268009.1268025.

[17] Sabin Devkota, Matthew P LeGendre, Adam Kunen, Pascal Aschwanden, and Katherine E Isaacs. 2022. Domain-centered support for layout, tasks, and specification for control flow graph visualization. In *2022 Working Conference on Software Visualization (VISSOFT)*. IEEE, 40–50.

[18] Philippe Dugerdil and Sazzadul Alam. 2008. Execution trace visualization in a 3d space. In *Fifth International Conference on Information Technology: New Generations (itng 2008)*. IEEE, 38–43.

[19] Julien Fierz. 2009. *Compass: Flow-Centric Back-In-Time Debugging*. Master's thesis. University of Bern. https://scg.unibe.ch/archive/masters/Fier09a.pdf.

[20] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. 2013. Live trace visualization for comprehending large software landscapes: the explorviz approach. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 1–4. DOI: 10.1109/VISSOFT.2013.6650536.

[21] Thomas M. J. Fruchterman and Edward M. Reingold. 1991. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21, 11, 1129–1164. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380211102. DOI: 10.1002/spe.4380211102.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. (1st ed.). *Addison-Wesley Professional Computing Series*. Pearson Education. ISBN: 9780321700698.

[23] Paul Gestwicki and Bharat Jayaraman. 2005. Methodology and architecture of jive. In *Proceedings of the 2005 ACM Symposium on Software Visualization* (SoftVis '05). Association for Computing Machinery, St. Louis, Missouri, 95–104. ISBN: 1595930736. DOI: 10.1145/1056018.1056032.

[24] Brendan Gregg. 2016. The flame graph. *Commun. ACM*, 59, 6, (May 2016), 48–57. DOI: 10.1145/2909476.

Table 3: Performance evaluation of the TRACE4D prototype for different program traces with respect to frame rate, memory consumption, and saving/loading times. We measure the frame rate both during the initial force simulation and afterward when playing the animation of the program trace. We found the frame rate to be sufficient for most of the considered program traces but see the need for optimization for larger program traces with respect to trace serialization, force simulation, and 3D rendering.

	Backend (Squeak)				Frontend (THREE.js)			
Program	**Tracer [s]**	**Serializer [s]**	**Serialization [kB]**	**Start-up [s]**	**Frame rate (force simulation) [FPS]**	**Frame rate (animation) [FPS]**	**RAM [MB]**	**GPU [MB]**
Regex *engine*								
• Test	0.1	0.1	0.1	0.1	60	60	60	60

[25] A. Hamou-Lhadj and T. Lethbridge. 2006. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, 181–190. DOI: 10.1109/ICPC.2006.45.

[26] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. 2004. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research* (CASCON '04). IBM Press, Markham, Ontario, Canada, 42–55. DOI: 10.5555/1034914.1034918.

[27] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. 2006. Design and implementation of a backward-in-time debugger. In *NODe 2006 – GSEM 2006*. Robert Hirschfeld, Andreas Polze, and Ryszard Kowalczyk, (Eds.) Gesellschaft für Informatik e.V., Bonn, 17–32. DOI: 20.500.12116/24100.

[28] Adrian Hoff, Lea Gerling, and Christoph Seidl. 2022. Utilizing software architecture recovery to explore large-scale software systems in virtual reality. English. In *2022 Working Conference on Software Visualization (VISSOFT)*. IEEE, United States, (Oct. 2022). ISBN: 978-1-6654-8093-2. DOI: 10.1109/VISSOFT55257.2022.00020.

[29] 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* (OOPSLA '97) number 10. Vol. 32. Association for Computing Machinery, Atlanta, Georgia, USA, (Oct. 1997), 318–326. DOI: 10.1145/263700.263754.

[30] D.F. Jerding and J.T. Stasko. 1998. The information mural: a technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization and Computer Graphics*, 4, 3, 257–271. DOI: 10.1109/2945.722299.

[31] 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* (ICSE '08). Association for Computing Machinery, Leipzig, Germany, 301–310. ISBN: 9781605580791. DOI: 10.1145/1368088.1368130.

[32] Alexander Krause, Malte Hansen, and Wilhelm Hasselbring. 2021. Live visualization of dynamic software cities with heat map overlays. In *2021 Working Conference on Software Visualization (VISSOFT)*, 125–129. DOI: 10.1109/VISSOFT52517.2021.00024.

[33] J. B. Kruskal and J. M. Landwehr. 1983. Icicle plots: better displays for hierarchical clustering. *The American Statistician*, 37, 2, 162–168. eprint: https://www.tandfonline.com/doi/pdf/10.1080/00031305.1983.10482733. DOI: 10.1080/00031305.1983.10482733.

[34] Danny B. Lange and Yuichi Nakamura. 1997. Object-oriented program tracing and visualization. *Computer*, 30, 5, (May 1997), 63–70. DOI: 10.1109/2.589912.

[35] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. 2008. Exploring the evolution of software quality with animated visualization. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, 13–20. DOI: 10.1109/VLHCC.2008.4639052.

[36] François Lemieux and Martin Salois. 2006. Visualization techniques for program comprehensiona literature review. In *Proceedings of the 2006 Conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the Fifth SoMeT_06*. IOS Press, NLD, 22–47. ISBN: 1586036734. DOI: 10.5555/1565321.1565325.

[37] Bil Lewis. 2003. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*. Vol. cs.SE/0310016, 11 pages. DOI: 10.48550/ARXIV.CS/0310016.

[38] 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)* number 1. Vol. 35, 63–79. DOI: https://doi.org/10.1016/j.cl.2008.05.006.

[39] Adrian Lienhard, Julien Fierz, and Oscar Nierstrasz. 2009. Flow-centric, back-in-time debugging. In *Objects, Components, Models and Patterns: 47th International Conference, TOOLS EUROPE 2009, Zurich, Switzerland, June 29-July 3, 2009. Proceedings 47*. Springer, 272–288.

[40] Adrian Lienhard, Tudor Girba, Orla Greevy, and Oscar Nierstrasz. 2008. Test blueprints - exposing side effects in execution traces to support writing unit tests. In *2008 12th European Conference on Software Maintenance and Reengineering*, 83–92. DOI: 10.1109/CSMR.2008.4493303.

[41] 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, (July 2008), 592–615. ISBN: 978-3-540-70591-8. DOI: 10.1007/978-3-540-70592-5_25.

[42] Daniel Limberger, Willy Scheibel, Jürgen Döllner, and Matthias Trapp. 2019. Advanced visual metaphors and techniques for software maps. In *Proceedings of the 12th International Symposium on Visual Information Communication and Interaction* (VINCI '19) Article 11. Association for Computing Machinery, Shanghai, China, 8 pages. ISBN: 9781450376266. DOI: 10.1145/3356422.3356444.

[43] Daniel Limberger, Willy Scheibel, Jürgen Döllner, and Matthias Trapp. 2022. Visual variables and configuration of software maps. *J. Vis.*, 26, 1, (Sept. 2022), 249–274. DOI: 10.1007/s12650-022-00868-1.

[44] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. 2004. Visualizing programs with jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces* (AVI '04). Association for Computing Machinery, Gallipoli, Italy, 373–376. ISBN: 1581138679. DOI: 10.1145/989863.989928.

[45] Johann Mortara, Philippe Collet, and Anne-Marie Dery-Pinna. 2021. Visualization of object-oriented variability implementations as cities. In *2021 Working Conference on Software Visualization (VISSOFT)*. IEEE, 76–87.

[46] T. Munzner. 1997. H3: laying out large directed graphs in 3d hyperbolic space. In *Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)* (INFOVIS '97). IEEE Computer Society, USA, 2. ISBN: 0818681896.

[47] B. A. Myers. 1986. Visual programming, programming by example, and program visualization: a taxonomy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '86). Association for Computing Machinery, Boston, Massachusetts, USA, 59–66. ISBN: 0897911806. DOI: 10.1145/22627.22349.

[48] Kunihiro Noda, Takashi Kobayashi, Tatsuya Toda, and Noritoshi Atsumi. 2017. Identifying core objects for trace summarization using reference relations and access analysis. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1, 13–22. DOI: 10.1109/COMPSAC.2017.142.

[49] Wim De Pauw, Doug Kimelman, and John M. Vlissides. 1994. Modeling object-oriented program execution. In *Proceedings of the 8th European Conference on Object-Oriented Programming* (ECOOP '94). Springer-Verlag, Berlin, Heidelberg, 163–182. ISBN: 3540582029. DOI: 10.5555/646152.679378.

[50] Guillaume Pothier and Éric Tanter. 2009. Back to the future: omniscient debugging. *IEEE Software*, 26, 6, 78–85. DOI: 10.1109/MS.2009.169.

[51] Luc Prestin. 2022. Hidden modularity. Retrieved May 5, 2023 from https://github.com/LucPrestin/Hidden-Modularity.

[52] Steven P. Reiss. 2007. Visual representations of executing programs. *Journal of Visual Languages & Computing*, 18, 2, 126–148. Selected papers from Visual Languages and Computing 2005 (VLC '05). DOI: https://doi.org/10.1016/j.jvlc.2007.01.003.

[53] Steven P. Reiss. 2006. Visualizing program execution using user abstractions. In *Proceedings of the 2006 ACM Symposium on Software Visualization* (SoftVis '06). Association for Computing Machinery, Brighton, United Kingdom, 125–134. ISBN: 1595934642. DOI: 10.1145/1148493.1148512.

[54] George Robertson, David Ebert, Stephen Eick, Daniel Keim, and Ken Joy. 2009. Scale and complexity in visual analytics. *Information Visualization*, 8, 4, (Dec. 2009), 247–253. DOI: 10.1057/ivs.2009.23.

[55] 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, 26 pages. ISBN: 0130280917. https://rmod-files.lille.inria.fr/FreeBooks/CollectiveNBlueBook/Rowledge-Final.pdf.

[56] Leon Schweizer. 2014. *PathObjects: Revealing Object Interactions to Assist Developers in Program Comprehension*. Master's thesis. Hasso Plattner Institute, University of Potsdam. https://github.com/leoschweizer/PathObjects-Thesis.

[57] Ben Shneiderman and Catherine Plaisant. 2005. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. (4th ed.). Pearson Education, India. ISBN: 0-321-19786-0. http://seu1.org/files/level5/IT201/Book%20-%20Ben%20Shneiderman-Designing%20the%20User%20Interface-4th%20Edition.pdf.

[58] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education (TOCE)*, 13, 4, 1–64.

[59] Tarja Systä, Kai Koskimies, and Hausi Müller. 2001. Shimba—an environment for reverse engineering java software systems. *Software: Practice and Experience*, 31, 4, 371–394. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.386. DOI: https://doi.org/10.1002/spe.386.

[60] Alfredo R. Teyseyre and Marcelo R. Campo. 2009. An overview of 3d software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15, 1, 87–105. DOI: 10.1109/TVCG.2008.86.

[61] Christoph Thiede and Patrick Rein. 2023. *Squeak by Example*. Vol. 6.0. ISBN 978-1-4476-2948-1. Lulu, 328 pages. ISBN: 9781447629481. https://www.lulu.com/shop/patrick-rein-and-christoph-thiede/squeak-by-example-60/paperback/product-8vr2j2.html.

[62] 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* (<Programming> '23). ACM, Tokyo, Japan, (Mar. 2023), 7 pages. DOI: 10.1145/3594671.3594678.

[63] 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). To appear. ACM, Cascais, Portugal, (Oct. 2023), 14 pages. DOI: 10.1145/3622758.3622892.

[64] Danny Tramnitzke. 2007. *Object Call Graph Visualization*. Bachelor Thesis. Växjö University. https://www.diva-portal.org/smash/get/diva2:205514/FULLTEXT01.pdf.

[65] Jonas Trümper, Alexandru C Telea, and Jürgen Döllner. 2012. Viewfusion: correlating structure and activity views for execution traces. In *TPCG*. Citeseer, 45–52.

[66] 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.

[67] Jan Waller, Christian Wulf, Florian Fittkau, Philipp Döhring, and Wilhelm Hasselbring. 2013. Synchrovis: 3d visualization of monitoring traces in the city metaphor for analyzing concurrency. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 1–4. DOI: 10.1109/VISSOFT.2013.6650520.

[68] Richard Wettel and Michele Lanza. 2007. Visualizing software systems as cities. In (June 2007), 92–99. DOI: 10.1109/VISSOF.2007.4290706.

[69] Linda Woodburn, Yalong Yang, and Kim Marriott. 2019. Interactive visualisation of hierarchical quantitative data: an evaluation. In *2019 IEEE Visualization Conference (VIS)*, 96–100. DOI: 10.1109/VISUAL.2019.8933545.