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: Screenshot of an animated object map showing a program trace for the construction of a regular expression matcher in the Squeak/Smalltalk programming environment. Blocks represent objects, arrows display references between objects, and color highlights and trails show object activations. The timeline at the bottom provides a temporal overview of the program trace and allows users to control the animation.

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 to build a mental model of a system's behavior that connects abstract concepts to implementation artifacts. We propose a novel approach to 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 plays a crucial role in software development. Programmers often come across unfamiliar systems that they want to fix, change, or extend. For this, they need to build up a mental model that links 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, identifying relevant units and actors and exploring 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 nonlinear fashion [39, 28, 54, 43, 53]. However, omniscient

debuggers are unsuitable 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 various metaphors such as cities or forests have been found to be useful for program comprehension tasks [75, 2, 45]. Yet, most approaches neglect the dynamic behavior of systems and take a coarse-grained view of their structure. As a result, these maps are inadequate 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 [71].

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 object maps* (or *(animated) object maps* for short), which depict 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 using animated 2.5D object maps.
(2) We describe the implementation of our prototype TRACE4D that applies this approach using a program tracer from the 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, encompassing responsiveness, frame rates, and memory consumption, for different program traces.

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

The remainder of this paper is structured as follows: in section 2, we discuss related work on the visualization of software architecture and behavior. 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 explain 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 [50, 64, 62, 46], *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 [57, 11, 12, 18].

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.[2] as a nested set of shapes [44, 45]. They offer various 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 other polygons as in Voronoi tesselation treemaps [6]. One popular, contemporary type of treemaps is *2.5D treemaps* which add a third dimension to the visualization by transforming each shape into a right prism (usually a 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 [19, 75, 15, 1, 48, 29, 45].

Thematic software maps. Unlike treemaps which display the programmer-specified organization of a software system, *thematic software maps* are a type of *topic maps* that 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 layout [4]. Different metaphors have been proposed to embody these graphs in a map, including board games [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 [38, sec. 3.4]. Dynamic information can relate to the behavior or evolution of software: for example, EVOSPACES [19] highlights classes in a software city when they are activated, while DYNACITY [16], EXPLORVIZ [33], SYNCHROVIS [73], and others [13] also draw connections between modules to visualize dataflow between them; Langelier et al. [37] gradually construct a software city and update 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 [21] while others replay a previously recorded trace of software activity [19].

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 parts of a program's object graph [47, 24]. Some graphs mimic 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 [36, 26].

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

[2] aut cetera: or so on

Communication flow. *Call graphs* and *control-flow graphs* are two popular ways of displaying entities with their mutual dynamic interactions or communications. Entities can be nodes from an object graph [68], organizational units such as classes [56] or modules [55], or subject to user-selected abstraction levels [36, 17, 72]. Avid [72] and PathObjects [60] provide animated object graphs where users can explore the control flow interactively. Boothe et al. merge 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 [7].

In contrast to traditional call graphs, some works have proposed peripheral, hierarchical layouts of nodes such as Extravis' *circular bundle views* [14] or the H3 hyperbolic 3D layout [49], which provide better scaling for highly connected graphs.

Dataflow. Another perspective that can be taken on the object graph is how state is propagated 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 [32]. Lienhard et al. propose an *inter-unit flow view* that displays the amount of data or objects exchanged between different classes or modules in a directed weighted graph [40]; this graph can also be embedded into a traditional call graph [41].

State changes. Lienhard et al. also propose a *side-effects graph* [41, 20] (also referred to as *test blueprints* [42]), which shows connections between objects changing each other's state. Similarly, *object traces* describe a way to slice a call tree (section 2.3) for exploring the state evolution of individual objects [66]. *Memory cities* support the heap memory analysis of programs by displaying objects and their memory consumption in a 2.5D treemap and animating the allocation and deallocation of objects [74].

2.3 Time-Centric Behavior Visualization

Besides the communication or evolution of entities, another perspective that visualizations often take on software behavior is the temporal order of program execution.

Call trees. A call tree is a hierarchy of stack frames or method invocations that can be obtained 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* [34, 69, 76]. Similarly to icicle plots, *flame graphs* show the historical call stack over time but also assign colors to stack frames to display additional performance data from profiling tools [25].

Sequential displays. UML sequence diagrams are a traditional approach to displaying communication between objects over time. Several tools adopt [63] and extend [26] this diagram type: for example, ISVis' *information mural* [31] and Extravis' *massive sequence view* [14] derive miniaturized versions of a sequence diagram [38, sec. 3.4], and Ovation [17] detects execution patterns to reduce sequence diagrams [26].

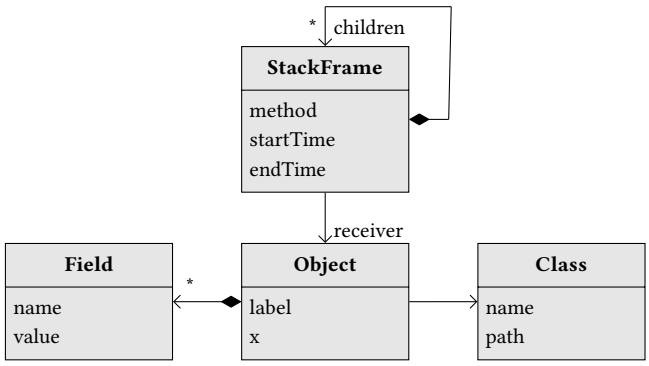

Figure 2: UML class diagram showing the data model of an object-oriented program trace for the visualization.

3 VISUALIZATION APPROACH

To support the comprehension of object-oriented programs, we propose *animated 2.5D object maps* as a novel visualization approach for program traces. In the following, we describe the prerequisites and the design of our 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 [67].

We assume a minimal data model of the program trace (fig. 2): the *call tree* is represented as a composite structure of *stack frames* 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 by 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 asynchronous or concurrent program behavior and metaprogramming peculiarities 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 2.5D 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* of objects, i.e., the invocation of any of their methods, and their *interaction*, i.e., the exchange of messages between two objects. They can navigate freely 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

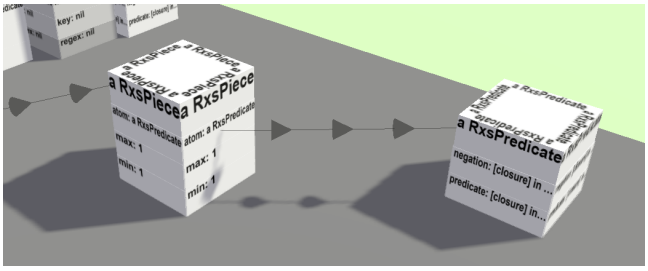

Figure 3: Visual mapping of objects, fields, and references to block entities, tiles, and arrows in the object map.

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 *tiles* 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 tile of the referencing field to the closest label of the referenced object's entity. To indicate the direction of arrows, 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. All object blocks are placed on a plane in the 2.5D object map. For their arrangement, we define a force-directed graph layout [22]. Between each pair of object blocks a and b, we apply several *weighted attractive forces* based on the class membership (F_{class}), the organizational proximity of classes (F_{org}), and the references (F_{ref}) and communication (F_{comm}) between objects. In the following definitions, the respective w denote the weight of each force, and $\text{org}(o)$ denotes the organizational path of an object o's class (e.g., a file path):

$$F_{\text{class}}(a,b) = \begin{cases} w_{\text{class}}, & \text{if } \text{class}(a) = \text{class}(b); \\ 0, & \text{otherwise.} \end{cases}.$$

$$F_{\text{org}}(a,b) = w_{\text{org}}\big(\text{commonPrefixLength}(\text{org}(a), \text{org}(b))\big)\,.$$

$$F_{\text{ref}}(a,b) = w_{\text{ref}}\big(\text{number of fields in } a \text{ that reference } b\big)\,.$$

$$F_{\text{comm}}(a,b) = w_{\text{comm}}\big(\text{number of messages from } a \text{ to } b\big)\,.$$

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.

Table 1: Default configuration of force weights for the object graph layout (columns represent assignments). References between objects dominate the layout while organizational proximity and communication between objects are weighted lower. Users can override these weights for specific program traces.

Object-specific forces				Generic forces	
w_{class}	w_{org}	w_{ref}	w_{comm}	w_{repulse}	w_{center}
0.001	$F \mapsto 0.005\,(\log_{10}(F)+1)$	0.1	0.00001	0.2	0.00142

We provide an empirical base configuration for all force weights but allow users to override them for specific program traces. By default, we give the highest weight to reference forces and the lowest weight to organizational forces with a six-order-of-magnitude difference and scale organizational forces logarithmically (table 1). This configuration encourages a state-centric layout of the object graph while leaving a margin for the characteristics of particular program traces (e.g., their ratio between intrinsic and extrinsic state [23, p. 218ff]) towards a more dataflow-driven layout. In addition, users can drag and drop blocks to customize the layout. To reduce response time [61, chap. 11] and maintain an experience of immediacy [70], we render the graph at regular update intervals even 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 filter system for excluding objects based on their label, class, or organization. Similar 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 indicates its recent activity: *inactive* blocks are colored in a neutral light gray while *active* blocks whose objects have recently received a message are highlighted in a bright red (fig. 4). After the control flow passes on to other objects, blocks fade linearly back to the base color within one second, thus applying a single-hue continuous sequential color scheme by Brewer et al.[3]

In addition to the color coding, a *trail* connects the $k = 15$ most recent object activations to support the delayed observation of short activations and the recognition of the exact activation order. The trail curve is based on a centripetal Catmull-Rom spline [9] with control points are placed on the top of each relevant block and alternated with intermediate points between blocks. Block control points are randomized using a normal distribution to distinguish multiple activations of the same object. Intermediate control points are raised vertically 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 during the animation 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 navigation aid. The timeline consists of two widgets stacked on top of each other (fig. 5): a *player* with a slider and a play/pause button displays the current point in time of the program trace and allows users to control the time and animation state. Behind the player, a collapsed *flame graph* shows the course of the call stack depth. Users can resize the timeline to explore the full call tree hierarchy and examine individual frames in the flame graph.

Both the flame graph and the object map are interactively linked; i.e., users can hover over an object in the map to discover all of its

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

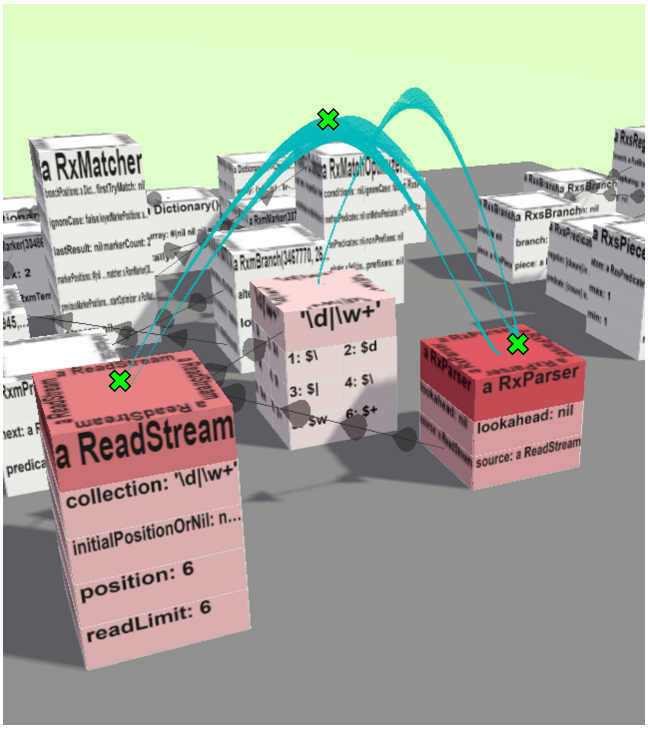

Figure 4: Visual mapping of object behavior to block colors and trail in the object map. The intensity of the red color indicates the recency of the last message received the corresponding object. The gradient trail curve connects the most recent object activations (control points of the curve are marked with a cross (✖)).

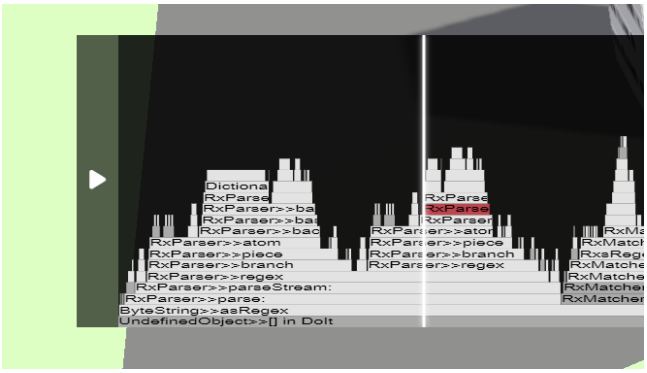

Figure 5: Timeline overlay consisting of widgets for controlling the playback of the program trace and a flame graph with a variable level of detail for navigating the call tree. The flame graph and the object map are interactively linked, e.g., the user can hover over a frame to highlight the corresponding object in the map.

activations in the timeline, or vice versa, they can click on a frame to fast-forward or rewind the trail in the map to the corresponding object activation. Thus, object map and timeline provide two orthogonal means of navigating through the object-oriented program trace at different granularities.

4 IMPLEMENTATION

We demonstrate the feasibility of animated 2.5D object maps by describing the implementation of our prototype TRACE4D that displays program traces from a Squeak/Smalltalk environment (*backend*) in a web application (*frontend*).

Program tracing. Squeak/Smalltalk is an interactive development environment (IDE) that is based on the object-oriented paradigm (everything is an object, including classes, methods, and stack frames) and gives programmers rich control to inspect and manipulate all parts of the system (by instrumenting method objects, recording stack frame objects, etc.) [30, 59, 65]. In our backend implementation, we use the TRACEDEBUGGER[4] [66], 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 [65, 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 example, the view on a dictionary will omit its internal overallocation array structure and instead display a more comprehensible collection of key-value pairs. As for the objects referenced as values from fields, we include in the serialization only those objects that receive at least one message in the program trace but only store a flat string representation of all other objects to avoid traversing the entire object graph of the system, most of which is irrelevant to the particular program trace.

Visualization. We implement the visualization frontend of TRACE-4D as a JavaScript web application. The web app retrieves a serialized program trace and provides a programmatic 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 at a configurable speed (defaulting to 50 bytecode instructions per second) and update the color highlights and trail for activated objects at each animation tick.

[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).

5 USE CASE: EXPLORING THE INTERNALS OF A REGULAR EXPRESSION ENGINE

To illustrate how animated object maps can support program comprehension, we describe how a fictional programmer could 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 better understanding of the subsystems involved 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 can discover several meaningful objects and clusters of objects:

- the pattern string `'\d|\w+'`;
- an `RxParser` object accessing the string via a `ReadStream`;
- eight objects referencing each other whose class names begin with the prefix `Rxs`, identifying them as nodes of the AST;
- an `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`.

After getting 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 cursor position 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, accessing the pattern through the `ReadStream` and talking to the AST nodes, presumably to initialize them.
(2) Next, the matcher becomes 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, the match optimizer is active, accessing the AST again and talking to the set.

Thus, our programmer was able to gain a first overview of the different parts of the `Regex` package and their collaboration to realize the construction of the matcher. Besides, she also could notice that almost 50% of the time was spent in the match optimizer. Without a closer idea of the role of this object, she might suspect this step to be a bottleneck in the construction and wonder if the optimization is optional and could be skipped for certain uses of the regular expression. To dive deeper into the implementation of the `Regex` package, she can expand the flame graph of the timeline, identify a few entry point methods of the objects that she finds most interesting (e.g., `RxParser»parseStream:`), and open them in the Squeak IDE to browse their source code.

[9]The interactive visualization of the described trace is available at https://linqlover.github.io/trace4d/app.html?trace=traces/regexParse.json and in the Wayback Machine of the Internet Archive. We also provide a screencast at https://github.com/LinqLover/trace4d/blob/main/assets/examples.md#program-trace-regexparsejson.

6 DISCUSSION

We discuss the potential and limitations of our visualization approach by reporting on our experiences and evaluating the performance of the TRACE4D prototype for six different use cases.

6.1 Experience Report

To assess 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 chose these criteria in view of short gulfs of execution and evaluation [52] and a maximum of information that users can gain from the visualization. We provide a full protocol of the experience report in appendix A.

Suitable traces. We had better experiences when using the visualization for smaller program traces such as various string processing examples. On the contrary, we were more challenged when 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 to be 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 homogeneous objects or unrelated subsystems contain more repetitive or irrelevant elements and are typically less amenable to exploration through animated object maps. Thus, it is the task of programmers to condense interesting behavior into a minimal 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.

Table 2: Ratings of our experience with animated object maps for program comprehension (appendix A). 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	○	+	+	+	+

Program comprehension. For suitable program traces, we were able to gain several kinds of insights and benefits from the visualization: we were able 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, compilation, and optimization stages in the same use case). Based on this overview, we could develop and refine our mental model of the explored system's functioning and connect it to particular classes and objects in their implementation. Furthermore, the interactive visualization helped us to explore and analyze communication patterns, reflect on the system design, and share and discuss our mental models with other developers.

Object graph layout. The structure of the object graph layout is critical 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 types 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 envision a clustering of objects into groups that could be displayed in a clearer structure through color coding or a hierarchical layout of objects. Clusters could be derived either from the existing force-directed layout or based on other distance metrics for objects such as their class organization, their communication patterns, or from embedding representations based on their source code or documentation.

Limitations. A general challenge in information visualization is to reduce the complexity of the underlying data to a comprehensible yet meaningful level [58]. For animated object maps, this challenge manifests as ourselves 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, configuration requires manual effort and is therefore a barrier for users to overcome. To reduce 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 an appropriate configuration for individual program traces.

Another source of complexity in animated object maps is the cluttered communication between different objects that is caused by lengthy handshakes between objects, messages that are irrelevant to the high-level program behavior, etc. To address this challenge, we aim to apply trace summarization techniques to eliminate implementation details from the underlying program trace [27, 51].

6.2 Evaluation of Performance

Another challenge is the technical performance of the visualization which affected our experience for larger program traces. We evaluate the performance of our prototype in terms of the speed of tracing and serializing program traces, the start-up time of the visualization, the rendering frame rates during the initial force simulation as well as when playing the animation of the program trace, and the memory consumption of the visualization (table 3).

To avoid network latency, we host the TRACE4D frontend locally using Node.js' built-in `http-server` module[10] and view the visualization in a Google Chrome browser on the same machine.

[10] `http-server` v14.1.1, node v16.8.0.

Table 3: Performance measurements of the TRACE4D prototype for different program traces with respect to frame rate, memory consumption, and the saving times and loading times. We measure the frame rate both during the initial force simulation and when playing the animation afterward. We find the performance to be practical 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.

Program	Backend (Squeak)[a,b]			Frontend (THREE.JS)[a,c]				
	Tracer[d] [s]	Serializer[d] [s]	Serialization file [kB]	Start-up [s]	Frame rate (force simulation) [FPS]	Frame rate (animation) [FPS]	RAM [MB]	GPU [MB]
`Regex` *engine*								
• Construction	76	3317	138	1247	21.7	58.6	224	479
• Matching	104	10 127	316	2250	15.1	34.0	386	491
`Morphic` *UI framework*								
• Event handling	159	31 725	365	1571	22.3	52.1	267	530
• Layouting	74	14 164	369	2998	10.2	23.4	279	645
`Inspector` tool initialization	147	18 044	616	4676	5.6	15.3	397	1805
HTML parsing	176	17 494	453	7089	18.1	34.6	458	2022

[a] System: Windows 10 64-bit 22H1, Intel Core i7-8550U 1.80GHz, 16GB RAM, Intel UHD Graphics 620 8GB.
[b] Backend: TRACEDEBUGGER 2022-12-29, Squeak 6.1Alpha #22599, OpenSmalltalk VM 202206021410.
[c] Frontend: THREE.JS r156, single-threaded, Chrome 117.0.5938.62 (inner window size: 1920 × 963).
[d] Excluding garbage collection.

To measure the frame rate, we modify the `stats.js` library[11] to record the number of frames per second (FPS) and report the average frame rate after rendering the scene for 30 seconds. We retrieve the memory consumption from the Chrome Task Manager for the tab of the TRACE4D prototype and for the GPU process; to estimate the effective GPU consumption of the visualization, we subtract the GPU consumption before starting the visualization from the later GPU consumption. To avoid distortions from the garbage collector, we launch the visualization of each trace from an empty browser tab, exclude the first two seconds from the frame rate measurements, log the minimum memory consumption from a 30-second interval for each measurement, and report the best average frame rate and memory consumption of three runs for each 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 to optimize the frame rate, graphics memory consumption, and saving and loading times of traces to improve user experience and scalability. Note that the limited frame rate during the force simulation is a deliberate trade-off to reduce the time until the layout stabilizes and the animation can be played.

To speed up the saving times and loading times, we see great potential for optimization in applying object filters in the backend before serializing the program trace. To improve the responsiveness of the visualization, we consider replacing the current force-simulation library `d3-force` with 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 by 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 AND FUTURE WORK

In this paper, we proposed a novel approach to visualizing the behavior of object-oriented programs through animated 2.5D object maps that depict particular objects and their interactions from a program trace. We described the visual design of our approach and implemented it in a prototypical web application that displays program traces from a Squeak/Smalltalk environment. We illustrated how programmers can use our tool to explore the behavior of object-oriented programs and found that, especially for smaller and coherent program traces, they can gain several insights into the structure of the object graph and the segments of the program behavior while larger and more redundant program traces still pose practical challenges regarding the configuration of the object map, the clarity of the object graph, irrelevant details in the communication between objects, and the performance of the visualization.

While we used program traces from Squeak/Smalltalk in our prototype, it is not limited to this environment but could easily be applied to other object-oriented languages for which fine-grained (i.e., non-sampling) program tracers exist, such as Java (e.g., using LIVERECORDER[12]), JavaScript (e.g., using DBUX[13]), or Python (e.g., using PYTRACE[14]).

Although we have already seen promising results in our experience report, a user study with a larger number of participants and program traces is needed to thoroughly evaluate the potential and limitations of animated object maps for program comprehension.

In future work, we intend to scale our approach to larger program traces by experimenting with different (hierarchical) layout approaches [35, 4], automatic configuration heuristics, and trace summarization techniques [27, 51] as well as by investing in technical optimizations of our prototype. To further improve the compactness and overview of larger object graphs, we also consider defining state filters for hiding irrelevant fields of objects or defining conditions for aggregating similar objects into collapsed blocks.

Similarly, we hypothesize that the clarity of object graphs could benefit significantly from the use of domain-specific knowledge about the explored system [12]. For example, particular domains such as the `Regex` engine or the `Morphic` UI framework could provide meaningful labels for objects (e.g., "\d" instead of "a RxsPredicate"), recommended filter presets, or structural hints (e.g., suggesting the use of the variables `submorphs` and `owner` to display the composite structure of a Morphic widget tree in the object map).

To explore the full potential of animated object maps for programmers, we want to integrate them into their usual development process by embedding the visualization into an IDE such as the Squeak/Smalltalk environment. This would allow programmers to immediately [70] switch between the program trace visualization and existing code browsing and debugging tools to take different views on the system under exploration; for example, they could open an animated object map from an omniscient debugger, select an object in the map to inspect it in an inspector tool, click on a stack frame in the flame graph to browse it in an editor, or even monitor a running program in the visualization. In addition, this integration could improve the performance of our prototype, since data (e.g., the fields of filterable objects) could be streamed into the visualization on demand instead of serializing the entire trace in a single operation[15].

Finally, we see another interesting research direction in encoding additional information about the system under exploration into the visualization. For example, we could map domain-specific information or also static or dynamic software metrics to the color, size, or texture of object blocks to improve the recognizability and information of object regions or to make visible design quality issues or behavioral anomalies, resp. By going beyond simple blocks and mapping different objects to individual glyphs such as bridges, construction cranes, or vehicles, we could also apply several metaphors based on the design and communication patterns of systems or their underlying domain to further improve intuitive recognition and understanding [11, 77]. In particular, we could also make visible the evolution of the object graph by displaying and animating the historical state of objects [66, 67], adding and removing objects based on their lifecycle, and allowing programmers to navigate along the history of objects.

[11] https://mrdoob.github.io/stats.js/
[12] https://undo.io/
[13] https://domiii.github.io/dbux/
[14] https://pytrace.com/

[15] In terms of our TRACE4D prototype, one implementation strategy for this would be to embed the web app in Squeak using the MAGICMOUSE package (https://github.com/cmfcmf/MagicMouse) and exchange events and data through a WebSockets connection between the frontend and the backend.

ACKNOWLEDGMENTS

I sincerely thank Willy Scheibel for enabling and supervising this seminar project, providing me with inspiring and extensive insights into the field and methods of software visualization, and giving with valuable advice and feedback throughout the project.

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 de la 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 *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2021) - IVAPP*. INSTICC. SciTePress, 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 2.5D semantic software maps. In *Proceedings of the 17th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2022) - IVAPP*. INSTICC. SciTePress, 210–217. DOI: 10.5220/0010991100003124.

[4] Daniel Atzberger, Tim Cech, Willy Scheibel, Daniel Limberger, and Jürgen Döllner. 2023. Visualization of source code similarity using 2.5D 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. DOI: 10.1007/978-3-031-25477-2_8.

[5] Daniel Atzberger, Willy Scheibel, Daniel Limberger, and Jürgen Döllner. 2021. Software Galaxies: displaying coding activities using a galaxy metaphor. In *Proceedings of the 14th International Symposium on Visual Information Communication and Interaction* (VINCI '21) Article 18. Association for Computing Machinery, 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, 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*. Tessellations Publishing, 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* (SIGGRAPH '84) number 3. Vol. 18. Association for Computing Machinery, 177–186. DOI: 10.1145/800031.808596.

[9] Edwin Catmull and Raphael Rom. 1974. A class of local interpolating splines. In *Computer Aided Geometric Design*. Academic Press, 317–326. DOI: 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, 73–74. 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, 807–810. 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*. Springer International Publishing, 102–121. DOI: 10.1007/978-3-319-11245-9_6.

[13] Marcus Ciolkowski, Simon Faber, and Sebastian von Mammen. 2017. 3D 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, 189–198. 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] Tommaso Dal Sasso, Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. Blended, not stirred: multi-concern visualization of large software systems. In *3rd IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 106–115. DOI: 10.1109/VISSOFT.2015.7332420.

[16] 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: 10.1016/j.infsof.2022.106989.

[17] 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, 16. https://www.usenix.org/legacy/publications/library/proceedings/coots98/full_papers/depauw/depauw.pdf.

[18] 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. eprint: 2108.03047. DOI: 10.1109/VISSOFT55257.2022.00013.

[19] Philippe Dugerdil and Sazzadul Alam. 2008. Execution trace visualization in a 3D space. In *Proceedings of the Fifth International Conference on Information Technology: New Generations* (ITNG '08). IEEE. IEEE Computer Society, 38–43. DOI: 10.1109/ITNG.2008.137.

[20] Julien Fierz. 2009. *Compass: Flow-centric back-in-time debugging*. Master's thesis. University of Bern. https://scg.unibe.ch/archive/masters/Fier09a.pdf.

[21] 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.

[22] 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.

[23] Erich Gamma, Richard Helm, Ralpha Johnson, and John Vlissides. 1994. *Design patterns: elements of reusable object-oriented software*. (1st ed.). *Addison-Wesley Professional Computing Series*. Pearson Education. ISBN: 9780321700698.

[24] 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, 95–104. DOI: 10.1145/1056018.1056032.

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

[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, 42–55. https://users.encs.concordia.ca/~abdelw/papers/CASCON04-TraceToolSurvey.pdf.

[27] Abdelwahab Hamou-Lhadj and Timothy C. 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.

[28] 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. Gesellschaft für Informatik e.V., 17–32. ISBN: 978-3-88579-182-9. https://dl.gi.de/items/ed80bc70-fcdd-4899-acc2-57f6de552aba.

[29] 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. DOI: 10.1109/VISSOFT55257.2022.00020.

[30] 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, 318–326. DOI: 10.1145/263700.263754.

[31] Dean F. Jerding and John 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.

[32] 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, 301–310. DOI: 10.1145/1368088.1368130.

[33] 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.

[34] Joseph B. Kruskal and James 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.

[35] Adrian Kuhn, Peter Loretan, and Oscar Nierstrasz. 2008. Consistent layout for thematic software maps. In *2008 15th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 209–218. DOI: 10.1109/WCRE.2008.45.

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

[37] 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.

[38] François Lemieux and Martin Salois. 2006. Visualization techniques for program comprehension - a literature review. In *Proceedings of the 2006 Conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the Fifth SoMeT_06* (Frontiers in Artificial Intelligence and Applications). Vol. 147. IOS Press, 22–47. ISBN: 1586036734. https://ebooks.iospress.nl/volumearticle/3136.

[39] 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.

[40] 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: 10.1016/j.cl.2008.05.006.

[41] 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.

[42] 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.

[43] 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, 592–615. DOI: 10.1007/978-3-540-70592-5_25.

[44] 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, 8 pages. DOI: 10.1145/3356422.3356444.

[45] Daniel Limberger, Willy Scheibel, Jürgen Döllner, and Matthias Trapp. 2022. Visual variables and configuration of software maps. *Journal of Visualization*, 26, 1, 249–274. DOI: 10.1007/s12650-022-00868-1.

[46] Leonel Merino, Mohammad Ghafari, and Oscar Nierstrasz. 2018. Towards actionable visualization for software developers. *Journal of Software: Evolution and Process*, 30, 2, e1923. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1923. DOI: https://doi.org/10.1002/smr.1923.

[47] 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, 373–376. DOI: 10.1145/989863.989928.

[48] 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. IEEE Computer Society, 76–87. DOI: 10.1109/VISSOFT52517.2021.00017.

[49] Tamara 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, 2. ISBN: 0818681896. https://graphics.stanford.edu/papers/h3/.

[50] Brad 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). Vol. 17. Association for Computing Machinery, 59–66. DOI: 10.1145/22627.22349.

[51] 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.

[52] Donald A. Norman. 1986. Cognitive engineering. In *User-centered System Design: New Perspectives on Human-computer Interaction*. Lawrence Erlbaum Associates, 31–61. ISBN: 9780367807320. https://www.researchgate.net/profile/Donald-Norman-3/publication/235616560_Cognitive_Engineering/links/0c960536c18209b825000000/Cognitive-Engineering.pdf.

[53] Michael Perscheid, Michael Haupt, Robert Hirschfeld, and Hidehiko Masuhara. 2012. Test-driven fault navigation for debugging reproducible failures. *Information and Media Technologies*, 7, 4, 1377–1400. DOI: 10.11185/imt.7.1377.

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

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

[56] 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: 10.1016/j.jvlc.2007.01.003.

[57] 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, 125–134. DOI: 10.1145/1148493.1148512.

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

[59] Tim Rowledge. 2001. A tour of the Squeak object engine. In *Squeak: Open Personal Computing and Multimedia*. (1st ed.). Prentice Hall PTR, 26 pages. ISBN: 0130280917. https://rmod-files.lille.inria.fr/FreeBooks/CollectiveNBlueBook/Rowledge-Final.pdf.

[60] 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.

[61] Ben Shneiderman and Catherine Plaisant. 2005. *Designing the user interface: strategies for effective human-computer interaction*. (4th ed.). Pearson Education. ISBN: 0-321-19786-0. http://seu1.org/files/level5/IT201/Book%20-%20Ben%20Shneiderman-Designing%20the%20User%20Interface-4th%20Edition.pdf.

[62] 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, Article 15, 1–64. DOI: 10.1145/2490822.

[63] 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: 10.1002/spe.386.

[64] 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.

[65] 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.

[66] 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, 7 pages. DOI: 10.1145/3594671.3594678.

[67] 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, 14 pages. DOI: 10.1145/3622758.3622892.

[68] Danny Tramnitzke. 2007. *Object call graph visualization*. Bachelor Thesis. Växjö University. https://www.diva-portal.org/smash/get/diva2:205514/FULLTEXT01.pdf.

[69] Jonas Trümper, Alexandru C. Telea, and Jürgen Döllner. 2012. ViewFusion: correlating structure and activity views for execution traces. In *Theory and Practice of Computer Graphics*. The Eurographics Association, 45–52. DOI: 10.2312/LocalChapterEvents/TPCG/TPCG12/045-052.

[70] David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the experience of immediacy. *Commun. ACM*, 40, 4, 38–43. DOI: 10.1145/248448.248457.

[71] Anneliese von Mayrhauser and A. Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer*, 28, 8, 44–55. DOI: 10.1109/2.402076.

[72] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. 1998. Visualizing dynamic software system information through high-level models. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '98) number 10. Vol. 33. Association for Computing Machinery, 271–283. DOI: 10.1145/286936.286966.

[73] 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.

[74] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Memory cities: visualizing heap memory evolution using the software city metaphor. In *8th Working Conference on Software Visualization (VISSOFT)*, 110–121. DOI: 10.1109/VISSOFT51673.2020.00017.

[75] Richard Wettel and Michele Lanza. 2007. Visualizing software systems as cities. In *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE Computer Society, 92–99. DOI: 10.1109/VISSOF.2007.4290706.

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

[77] Hannes Würfel, Matthias Trapp, Daniel Limberger, and Jürgen Döllner. 2015. Natural phenomena as metaphors for visualization of trend data in interactive software maps. In *Computer Graphics and Visual Computing (CGVC)*. The Eurographics Association. DOI: 10.2312/cgvc.20151246.

A PROTOCOL OF EXPERIENCE REPORT

Here we provide the raw data of our experience report in subsection 6.1.

A.1 Criteria

(1) *Configuration effort*: How many operations are required to achieve a usable configuration in terms of filters and forces?
(2) *Clarity of objects*: Is the quantity of displayed objects manageable or overwhelming?
(3) *Object layout*: Is it possible and easy to identify regions of the object graph? How meaningful are the identified patterns?
(4) *Animation*: Is it possible and easy to recognize, follow, and perceive the flow of activity?
(5) *Program comprehension*: Is it possible and easy to identify sections of the program execution? How meaningful are the identified patterns?

A.2 Ratings

A.2.1 `Regex` *engine.*

Construction. Program trace: `regexParse.json`.

Criterion	Positive	Negative	Rating
Configuration effort	• no additional configuration required	-	+
Clarity of objects (26)	-	-	+
Object layout	• identified groups: input, AST, NFA	-	+
Animation	• manageable, overly consistent speed • no noise	• long delay in hidden dictionaries of `RxMatchOptimizer`	+
Program comprehension	• identified sections: parsing, compilation, optimization	-	+

Matching. Program trace: `regexMatch.json`.

Criterion	Positive	Negative	Rating
Configuration effort	• no additional configuration required	• wait a few seconds for force simulation to stabilize	+
Clarity of objects (31)	-	• too many similar input characters	+
Object layout	• identified groups: input, NFA	-	+
Animation	• overly consistent speed • no noise	• too lengthy animation / too slow speed	+
Program comprehension	• identified single matches	-	+

A.2.2 `Morphic` *UI framework.*

Event handling. Program trace: `mouseDown.json`.

Criterion	Positive	Negative	Rating

Configuration effort	-	• required force weights:
		`globalFactor = 0.1`
		• required object filters:
		`excludedClassNames.push('MorphExtension', 'Dictionary')`
		• wait many seconds for force simulation to stabilize
			–

Clarity of objects (53)	-	• too many morphs, too many events with redundant state • many irrelevant fields in morphs • labels too small when zooming out, poor font quality • did not find central button morph	–
Object layout	• identified groups: kinds of morphs, events	• overwhelmingly large cluster of morphs	○
Animation	• followable	• too lengthy animation • some delays in hidden objects	○
Program comprehension	• identified sections: event dispatching	• not identified receiver morph for event	○

Layouting. Program trace: `fullBoundsTextView.json`.

Criterion	Positive	Negative	Rating
Configuration effort	-	• required force weights: `globalFactor = 0.1`	○
Clarity of objects (29)	-	• many irrelevant fields in morphs • labels too small when zooming out	○
Object layout	• identified groups: central morphs, peripheral state	-	+
Animation	• followable	• too lengthy animation	○
Program comprehension	• identified sections: very rough tree traversal	• not understood tree structure or its implications on layout	−

A.2.3 *Inspection tool initialization.* Program trace: `inspectorResetFields.json`.

Criterion	Positive	Negative	Rating

Configuration effort	-	• required force weights:
		`repulsion = .4`
		`communication = 0`
		`globalFactor = .4`
		• required object filters:
		`excludedObjectNames.push('an Object')`
		• wait many seconds for force simulation to stabilize
			−

Clarity of objects (46)	-	• too many homogeneous inspector fields	−
Object layout	• identified groups: inspector fields, streams, texts	• cannot distinguish versions of inspector fields	−
Animation	-	• too lengthy animation • insufficient frame rate • some delays in hidden objects	−
Program comprehension	• identified sections: very rough traversal of inspector fields	• not identified different versions / traversals of inspector fields • not identified object under inspection	−

A.2.4 *HTML parsing.* Program trace: `asTextFromHtml.json`.

Criterion	Positive	Negative	Rating

Configuration effort	-	• required object filters:
		`excludedClassNames.push('ByteString', 'Character')`
		• cannot filter on single relevant string
		• required player configuration:
		`player.stepsPerSecond = 200`
			○

Clarity of objects (21)	-	-	+
Object layout	• identified groups: parser with stack, text parts, streams	-	+
Animation	• followable	• slightly too lengthy animation	+
Program comprehension	• identified sections: parsing of HTML tags, pushing / popping of stack, construction of text parts	-	+