# Exposé: Supporting The Exploration of Program Traces through Animated Software Cities

Christoph Thiede

## 1 Background

### 1.1 Omniscient Debugging

*Omniscient debugging*, also referred to as *back-in-time debugging* or *time-travel debugging*, is a debugging technique that records the execution of a program and allows programmers to explore it independently of the original order of execution [18,26,34]. The resulting *program trace* consists of the program's behavior (typically represented through a *call tree* [29]) and, optionally, its historical states [29,44]. Programmers use omniscient debugging not only for fault localization but also for general *program exploration* as part of an *exploratory programming* process [36,38], e.g., to learn about the implementation of methods and the communication between objects in context, to refine their knowledge of an architecture, or to answer specific questions about the behavior of a system.

### 1.2 Program Visualization

*Program visualization* is a broad field of approaches that use graphical representations of individual program execution instances to support the understanding of programs [33,42,43]. Program visualization employs various visual variables such as text, shapes, iconography, layouts, or color to display the state or the behavior of programs. In particular, the term *algorithm animation* is mostly used in an educational context and focuses on interactive or moving visualizations of single algorithms [6]. Some program visualizations are entailed to specific domains or algorithms while others attempt to provide general-purpose mechanisms for visualizing arbitrary control flows or interactions [7,8,13,37].

For example, arbitrary program traces can be visualized by circular bundle views [10], flame graphs [45], or treemaps of the call tree [5]. Some approaches can also visualize running programs instead of completed program traces [15,17]. Specifically for *object-oriented systems*, there are several approaches to visualizing program traces that take different perspectives such as the communication between objects [11,40] or modules [35], their data flow [27,28], or their state changes [39,44].

### 1.3 Software Maps

*Software maps* are 3D visualizations that represent the structure of software projects [30]. Typical visual variables of software maps include the labels, shape and extent, color, and layout of entities. The structure of software maps is usually derived from the static organization of a project into a package or file hierarchy, while the layout can be based on the dependencies or similarities [3] between units. In addition, they can display further attributes of the software units such as metrics about their architectural quality or evolution [2]. A common type of software map is a *2.5D treemap* that resembles planar 2D treemaps but is extended with a third dimension for the height of the cuboid entities that displays a metric such as the number of methods for each unit. There are several other visual metaphors for representing treemaps, including landscapes [3], tessellations [4], and cities [1,19,30,32].

Software maps can also be enriched with *dynamic* information from program traces such as the activity in entities [23,48] or the amount of communication between them [9,12,15]. Some approaches also make the time of the underlying information accessible by

animating software maps [14,17,24,47] or providing means to navigate through them along the program execution [15:8.4.5,20].

Other approaches use the city metaphor to visualize a software's memory heap instead of its architecture [49]. Treemaps can also be combined with other types of visualizations such as graphs [51].

### 1.4 3D Programming Environments

*3D programming environments* enable programmers to explore software systems in a 3D space that contains program artifacts such as classes or methods [16,22,31]. These artifacts can be represented by normal code [16] or by visualizations such as software cities [31]. Programmers can interact with and navigate through the space using a conventional workstation [22] or a VR headset [16,31]. The arrangement of the representations may be static or customizable by programmers, and programmers may be able to modify the code from the 3D environment [16].

## 2 Research Question

Both program animation and software maps have proven to be useful visualizations for helping programmers explore programs. However, due to the limited space and the lack of intuitive metaphors, existing program animations are usually limited to simple algorithms or processes with a small number of entities. On the other hand, software maps mainly take a global and static or semi-static perspective on software architectures which is not suitable for representing concrete processes or data.

To bridge this gap, in this seminar project on *Methods and Techniques for Visual Analytics*, we want to examine the following question:

> How can we improve the exploration of program traces through animated software cities that reveal the relevant entities from the program execution?

Thus, we want to transfer the concepts of animated software maps for surveying large software systems from a static perspective to the in-depth exploration of subsystems in the context of concrete program traces.

To answer this question, we want to build a novel prototype and use it to explore program traces from different scenarios. Depending on the time available, we will also consider conducting a qualitative study ($N \leq 5$) to get feedback and experience reports from users.

## 3 Approach

We want to develop a prototype that provides programmers with an interactive 3D visualization of the relevant entities from a recorded program trace (see fig. 1):

- *Entities* can represent individual object instances, classes, or higher-level organizational units such as packages. They can display the protocols and methods of classes and the internal state of objects.
- The visualization is *animated*: programmers can use play/pause buttons or a time slider to travel through the time, and entities display the current activity (e.g., activated methods or changed variables) through visual variables (e.g., a bright color).
- The visualization is *customizable*: programmers can control the layout and positioning of entities, filter the displayed entities and adjust their granularity/clustering (e.g., bundle objects into a single class entity or explode a class entity into individual objects).
- The visualization is *interactive*: programmers can select entities of interest to view their activation times in the time slider or to browse their implementation in an IDE.

## 4 Implementation

To retrieve the program traces, we will use the TraceDebugger[1], an omniscient debugger for the

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

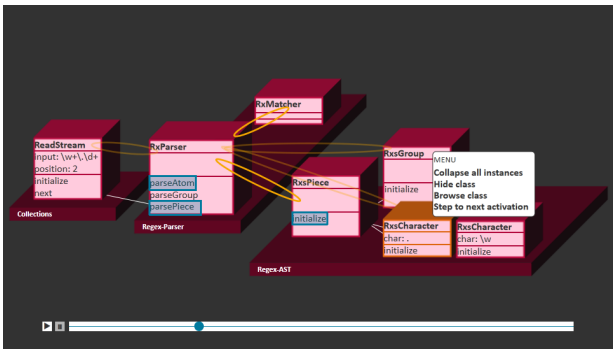


Figure 1: Wireframe of exploring the parsing process of a regular expression.

interactive programming system Squeak/Smalltalk[2] that provides an interface for tracing the behavior and state of a program. It also provides several small to medium-sized examples of program traces from different domains such as UI systems, parsing, and exception handling.

To create the interactive visualizations, we will use a modern high-level visualization framework such as Three.js or D3.js or a game engine such as Godot or Babylon.js.

To connect the backend and the frontend, we will export the program trace in a format such as JSON. Depending on the available time and focus, we may connect the backend and the frontend directly via WebSocket or long polling to enable programmers to navigate from their IDE to the visualization and back and to allow the visualization to dynamically retrieve additional data from the program trace. The visualization can be displayed in a web browser window next to the Squeak IDE, but it would also be possible to embed it into Squeak using a web browser proxy such as MagicMouse[3].

For building the prototype, we consider the following challenges and possible solution approaches:

**Geometric layout.** To decide the position of entities, we want to use a force-directed graph. For computing the forces, we want to explore a combination of the static structure of the class organization and dynamic information such as the frequency of messages exchanged between objects and their mutual references. We also want to enable the user to override the initial layout by manually moving entities to relax the need for an optimal layout algorithm and to respect the specific domain knowledge of the user, which is a common practice in exploratory programming environments that support direct manipulation.

**Linear time.** Because of method calls, recursion, and loops, call trees are deeply hierarchical and repetitive, but to play an animation, we need to condense the time into a linear view. Depending on the program trace, the condensed linear time may provide an inconsistent information density and prevent users from following the animated program execution smoothly. One way to address this issue would be to provide speed weights based on the depth of the call stack or user-provided inputs per unit. Users could also be provided with "step into"/"step over" buttons next to the time slider to skip redundant parts of the program trace. Ultimately, users could access the entire call tree for navigation, but this would increase the visual complexity and decrease usability.

**Filtering/clustering interface.** We will try to implement the filtering and clustering mechanisms within the visualization following a direct manipulation style. For example, users could left-click on an entity to explode it, or right-click on a component to collapse it with all its siblings. However, if the required gesture set lacks intuitiveness or becomes too complex, we could switch to a conventional interface like a multi-select list widget instead.

## 5 Next Steps

To bootstrap the prototype, we will design a serialization format for program traces from the TraceDebugger and implement a serializer in Squeak. We will finalize the choice of the visualization framework

[2] https://squeak.org/
[3] https://github.com/cmfcmf/MagicMouse

and get some first hands-on experience with it by creating a minimal representation of cuboids for the program trace. After that, we will explore further visual variables, add animation, and design means for filtering and clustering entities. For example, visual variables could be mapped to dynamic software metrics, tails between entities could display the current call stack, or users could select some state of an object to highlight its changes in the time slider.

## 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:https://doi.org/10.1109/VISSOFT52517.2021.00028

[2] 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.

[3] 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.

[4] 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, New York, NY, USA, 165–172. DOI:https://doi.org/10.1145/1056018.1056041

[5] Christoph Bockisch, Marnix van 't Riet, Haihan Yin, Mehmet Aksit, Ziyi Lin, Yuting Chen, and Jianjun Zhao. 2015. Trace-based debugging for advanced-dispatching programming languages. In *Proceedings of the 10th workshop on implementation, compilation, optimization of object-oriented languages, programs and systems* (ICOOOLPS '15), Association for Computing Machinery, New York, NY, USA. DOI:https://doi.org/10.1145/2843915.2843922

[6] 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.

[7] 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, New York, NY, USA, 807–810. DOI:https://doi.org/10.1145/1449814.1449869

[8] 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, Cham, 102–121. DOI:https://doi.org/10.1007/978-3-319-11245-9_6

[9] 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*, 189–198.

[10] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie Van Deursen, and Jarke J Van Wijk. 2008. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software* 81, 12 (2008), 2252–2268.

[11] Ward Cunningham and Kent Beck. 1986. A diagram for object-oriented programs. In *Conference proceedings on object-oriented programming systems, languages and applications* (OOPSLA '86), Association for Computing Machinery, New York, NY, USA, 361–367. DOI:https://doi.org/10.1145/28697.28734

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

[13] 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.

[14] 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.

[15] 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:https://doi.org/10.1109/VISSOFT.2013.6650536

[16] Leonard Geier, Clemens Tiedt, Tom Beckmann, Marcel Taeumel, and Robert Hirschfeld. 2022. Toward a VR-native live programming environment. In *Proceedings of the 1st ACM SIGPLAN international workshop on programming abstractions and interactive notations, tools, and environments*, 26–34.

[17] Orla Greevy, Michele Lanza, and Christoph Wysseier. 2006. Visualizing live software systems in 3D. In *Proceedings of the 2006 ACM symposium on software visualization*, 47–56.

[18] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. 2006. Design and implementation of a backward-in-time debugger. In *NODe 2006 – GSEM 2006*, Gesellschaft für Informatik e.V., Bonn, 17–32. DOI:https://doi.org/20.500.12116/24100

[19] Adrian Hoff, Lea Gerling, and Christoph Seidl. 2022. Utilizing software architecture recovery to explore large-scale software systems in virtual reality. In *2022 working conference on software visualization (VISSOFT)*, IEEE, 119–130.

[20] Akihiro Hori, Masumi Kawakami, and Makoto Ichii. 2019. Codehouse: Vr code visualization tool. In *2019 working conference on software visualization (VISSOFT)*, IEEE, 83–87.

[21] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. 2002. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing* 13, 3 (2002), 259–290.

[22] Pooya Khaloo, Mehran Maghoumi, Eugene Taranta, David Bettner, and Joseph Laviola. 2017. Code park: A new 3D code visualization tool. In *2017 IEEE working conference on software visualization (VISSOFT)*, 43–53. DOI:https://doi.org/10.1109/VISSOFT.2017.10

[23] 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:https://doi.org/10.1109/VISSOFT52517.2021.00024

[24] 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:https://doi.org/10.1109/VLHCC.2008.4639052

[25] Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A Uronen. 2003. The jeliot 2000 program animation system. *Computers & Education* 40, 1 (2003), 1–15.

[26] Bil Lewis. 2003. Debugging backwards in time. In *Proceedings of the fifth international workshop on automated debugging (AADEBUG 2003)*. DOI:https://doi.org/10.48550/ARXIV.CS/0310016

[27] 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)*, 63–79. DOI:https://doi.org/https://doi.org/10.1016/j.cl.2008.05.006

[28] 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.

[29] 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), Springer Verlag, Paphos, Cyprus, 592–615. DOI:https://doi.org/10.1007/978-3-540-70592-5_25

[30] Daniel Limberger, Willy Scheibel, Jürgen Döllner, and Matthias Trapp. 2023. Visual variables and configuration of software maps. *Journal of Visualization* 26, 1 (2023), 249–274.

[31] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. 2017. CityVR: Gameful software visualization. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*, IEEE, 633–637.

[32] 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.

[33] 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, New York, NY, USA, 59–66. DOI:https://doi.org/10.1145/22627.22349

[34] Guillaume Pothier and Éric Tanter. 2009. Back to the future: Omniscient debugging. *IEEE Software* 26, 6 (2009), 78–85. DOI:https://doi.org/10.1109/MS.2009.169

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

[36] Patrick Rein and Robert Hirschfeld. 2018. The exploration workspace: Interleaving the implementation and use of plain objects in smalltalk. In *Companion proceedings of the 2nd international conference on the art, science, and engineering of programming* (Programming '18), Association for Computing Machinery, New York, NY, USA, 113–116. DOI:https://doi.org/10.1145/3191697.3214339

[37] 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, New York, NY, USA, 125–134. DOI:https://doi.org/10.1145/1148493.1148512

[38] David W. Sandberg. 1988. Smalltalk and exploratory programming. *SIGPLAN Not.* 23, 10 (October 1988), 85–92. DOI:https://doi.org/10.1145/51607.51614

[39] Rodrigo Schulz, Fabian Beck, Jhonny Wilder Cerezo Felipez, and Alexandre Bergel. 2016. Visually exploring object mutation. In *2016 IEEE working conference on software visualization (VISSOFT)*, IEEE, 21–25.

[40] Leon Schweizer. 2014. PathObjects: Revealing object interactions to assist developers in program comprehension. Master's thesis. Hasso Plattner Institute, University of Potsdam. Retrieved from https://github.com/leoschweizer/PathObjects-Thesis

[41] Dominik Seifert, Michael Wan, Jane Hsu, and Benson Yeh. 2022. Dbux-PDG: An interactive program dependency graph for data structures and algorithms. In *2022 working conference on software visualization (VISSOFT)*, 141–151. DOI:https://doi.org/10.1109/VISSOFT55257.2022.00022

[42] 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 (2013), 1–64.

[43] Alfredo R. Teyseyre and Marcelo R. Campo. 2009. An overview of 3D software visualization. *IEEE Transactions on Visualization and Computer Graphics* 15, 1 (2009), 87–105. DOI:https://doi.org/10.1109/TVCG.2008.86

[44] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. Object-centric time-travel debugging: Exploring traces of objects. In *Proceedings of companion proceedings of the 7th international conference on the art, science, and engineering of programming* (<Programming> '23 companion), ACM, New York, NY, USA. Retrieved from https://2023.programming-conference.org/details/px-2023-papers/9/Object-centric-Time-Travel-Debugging-Exploring-Traces-of-Objects

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

[46] J Ángel Velázquez-Iturbide, Antonio Pérez-Carrasco, and Jaime Urquiza-Fuentes. 2008. SRec: An animation system of recursion for algorithm courses. *Acm sigcse bulletin* 40, 3 (2008), 225–229.

[47] 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. *ACM SIGPLAN Notices* 33, 10 (1998), 271–283.

[48] 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:https://doi.org/10.1109/VISSOFT.2013.6650520

[49] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Memory cities: Visualizing heap memory evolution using the software city metaphor. In *2020 working conference on software visualization (VISSOFT)*, IEEE, 110–121.

[50] 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:https://doi.org/10.2312/cgvc.20151246

[51] Shengdong Zhao, M. J. McGuffin, and M. H. Chignell. 2005. Elastic hierarchies: Combining treemaps and node-link diagrams. In *IEEE symposium on information visualization, 2005. INFOVIS 2005.*, 57–64. DOI:https://doi.org/10.1109/INFVIS.2005.1532129