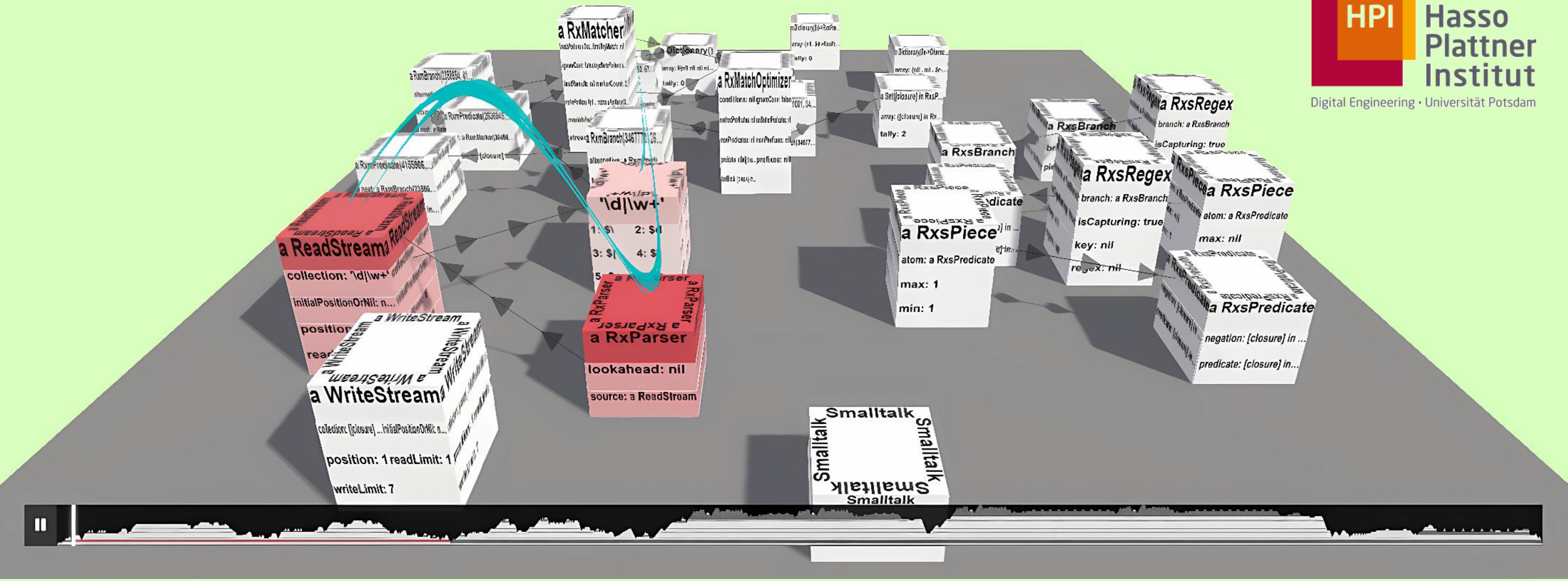


Hasso Plattner Institut
Digital Engineering · Universität Potsdam

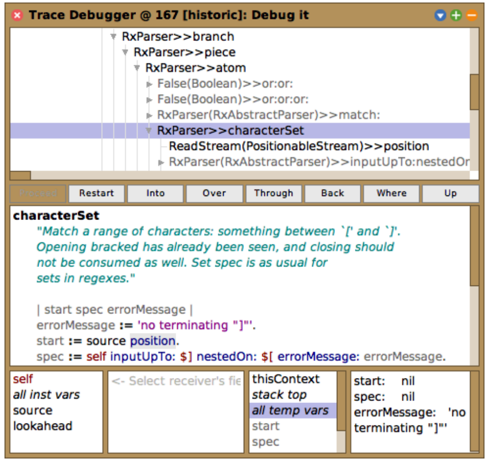
Bringing Objects to Life: Supporting Program Comprehension through Animated 2.5D Object Maps from Program Traces

IVAPP 2024
2024-02-27

Christoph Thiede, Willy Scheibel, Jürgen Döllner
Hasso Plattner Institute, Germany

Background

- **Increasing demand for program comprehension:**
  - Explore and **reverse-engineer** large **unfamiliar software systems**
  - **Build up a mental model** that links observable behavior to architectural and implementation units [Hamou-Lhadj et al., 2004; von Mayrhauser et al., 1995]
- **Traditional approaches:**
  - **Source-code centric exploration:** Package/file-based navigation
    - Abstract descriptions, overwhelming details
  - **Behavior-centric exploration:** Example-based navigation in a debugger
    - Observe **concrete program instances** (e.g., test cases) through their **call stack** to identify relevant **units and interactions**
    - **Omniscient debuggers** (or time-travel debuggers): Record a **program trace** to explore behavior in nonlinear fashion through a **call tree** [Pothier and Tanter, 2009; Perscheid et al., 2012]
    - Too fine-grained and text-heavy information displays



```
RxParser>>branch
  RxParser>>piece
    RxParser>>atom
      False(Boolean)>>or:or:
      False(Boolean)>>or:or:or:
      RxParser(RxAbstractParser)>>match:
      RxParser>>characterSet
        ReadStream(PositionableStream)>>position
        RxParser(RxAbstractParser)>>inputUpTo:nestedOr
```

```
characterSet
	"Match a range of characters: something between `[' and `]'.
	Opening bracked has already been seen, and closing should
	not be consumed as well. Set spec is as usual for
	sets in regexes."

	| start spec errorMessage |
	errorMessage := 'no terminating "]"'.
	start := source position.
	spec := self inputUpTo: $] nestedOn: $[ errorMessage: errorMessage.
```

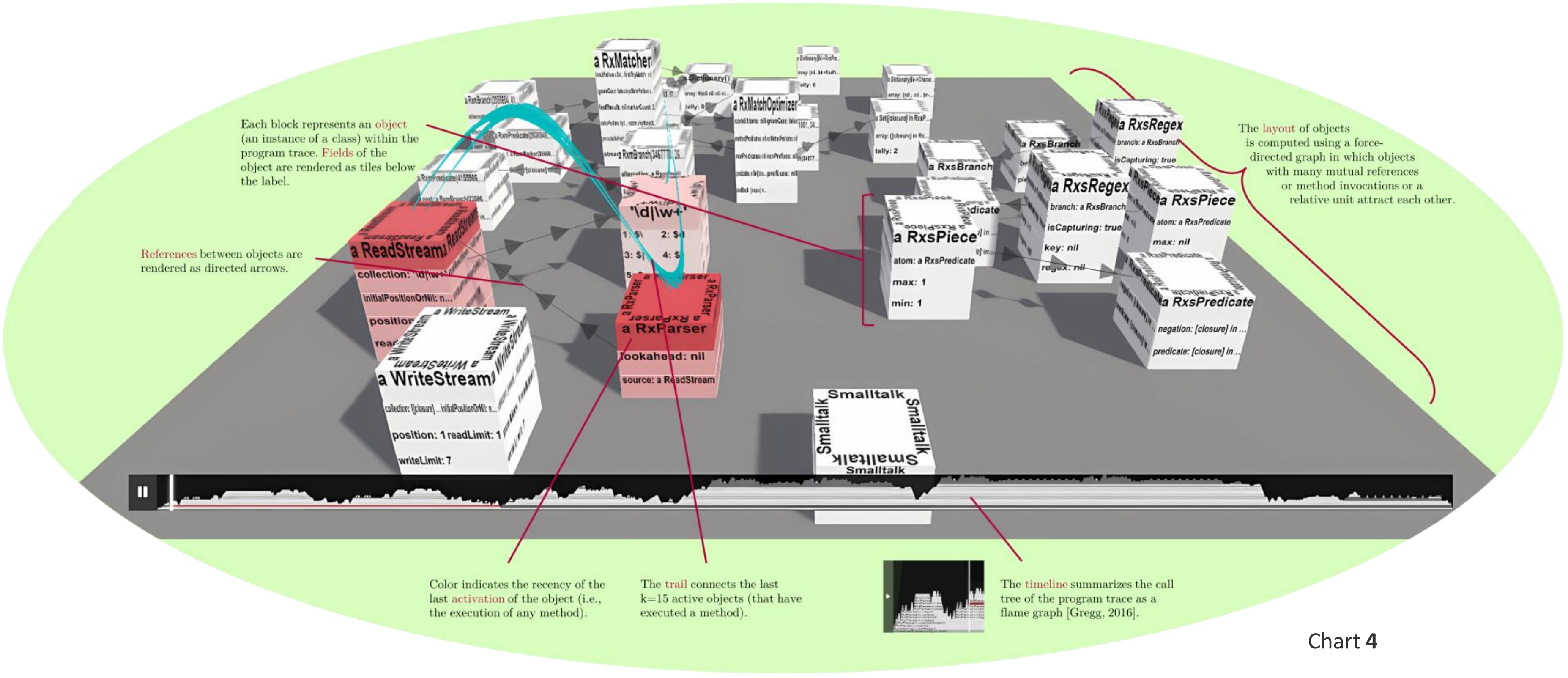
```
start:	nil
spec:	nil
errorMessage:	'no
terminating "]"'
```

Using the *TraceDebugger* in Squeak/Smalltalk to explore the parsing of a regular expression pattern. [Thiede et al., 2023a]

Approach

- Visualize **object-oriented programs** as **animated 2.5D object maps:**
  - **Object map: Blocks** represent objects, **arrows** represent references
  - **Force-directed graph layout** based on **classes, references, and interactions** between objects
  - **Animation: Object activations** and **communication** during program execution
    - Color indication [Harrower and Brewer, 2003] and curved trail (control flow)
  - **Timeline: Temporal navigation** and **call tree** overview [Gregg 2016]

Demo: The TRACE4D Prototype



Each block represents an object (an instance of a class) within the program trace. Fields of the object are rendered as tiles below the label.

References between objects are rendered as directed arrows.

The layout of objects is computed using a force-directed graph in which objects with many mutual references or method invocations or a relative unit attract each other.

Color indicates the recency of the last activation of the object (i.e., the execution of any method).

The trail connects the last k=15 active objects (that have executed a method).

The timeline summarizes the call tree of the program trace as a flame graph [Gregg, 2016].

Evaluation

- **Experience report of the TRACE4D prototype:**
  - Analyzed 6 program traces from 4 domains within the Squeak/Smalltalk environment
- **Suitable program traces:**
  - Small to moderate **number of subsystems and objects**
  - **High class cohesion** and **extensive communication** between objects
- **Types of visual insights:**
  - Identify **characteristic regions** of the **object graph**
  - Identify **significant behavior stages** within the program execution

Conclusion

- **Possible benefits for program comprehension:**
  - Develop and refine a **mental model** of system functioning and link it to **implementation artifacts**
  - Exploring and analyzing **communication patterns**
  - **Reflect** on system design and **share/discuss** mental models with coworkers
- **Future Work:**
  - **Visual scaling** for larger object graphs through **trace summarization** [Hamou-Lhadj and Lethbridge, 2006]
  - **Improve clarity** of the object graph through **clustering** and **hierarchical layout approaches** [Atzberger et al., 2023; Scheibel et al., 2018]
  - Evaluate potential and limitations through **user study**
  - **Integrate** visualization into programmers' toolchains

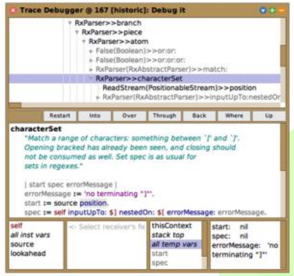
Bringing Objects to Life: Supporting Program Comprehension through Animated 2.5D Object Maps from Program Traces

Christoph Thiede, Willy Scheibel, and Jürgen Döllner
Hasso Plattner Institute, University of Potsdam, Germany

linqlover.github.io/trace4d

Abstract

Programmers who want to explore the architecture of software systems need appropriate visualizations such as software maps. However, existing software visualizations mainly display the static software structure, neglecting important dynamic runtime behavior. We propose *animated 2.5D object maps* that depict particular objects and their interac-tions from a program trace. From our experience of using our prototype with a couple of use cases, we conclude that animated 2.5D object maps can practically benefit program comprehension tasks, but further research is needed to improve scalability and usability.



```
Trace Debugger @ 167 [historic]: Debug it

▾ RxParser>>branch
  ▾ RxParser>>piece
    ▾ RxParser>>atom
      ▸ False(Boolean)>>or:or:
      ▸ False(Boolean)>>or:or:or:
      ▸ RxParser(RxAbstractParser)>>match:
      ▾ RxParser>>characterSet
          ReadStream(PositionableStream)>>position
        ▸ RxParser(RxAbstractParser)>>inputUpTo:nestedOr

[Restart] [Into] [Over] [Through] [Back] [Where] [Up]

characterSet
	"Match a range of characters: something between `[' and `]'.
	Opening bracked has already been seen, and closing should
	not be consumed as well. Set spec is as usual for
	sets in regexes."

	| start spec errorMessage |
	errorMessage := 'no terminating "]"'.
	start := source position.
	spec := self inputUpTo: $] nestedOn: $[ errorMessage: errorMessage.

self            <- Select receiver's fi   thisContext      start:  nil
all inst vars                             stack top        spec:   nil
source                                    all temp vars    errorMessage:  'no
lookahead                                 start            terminating "]"'
                                          spec
```

Figure 1: Using the TRACEDEBUGGER, an omniscient debugger for the Squeak/Smalltalk programming system, to explore the parsing of a regular expression pattern [Thiede, 2023].

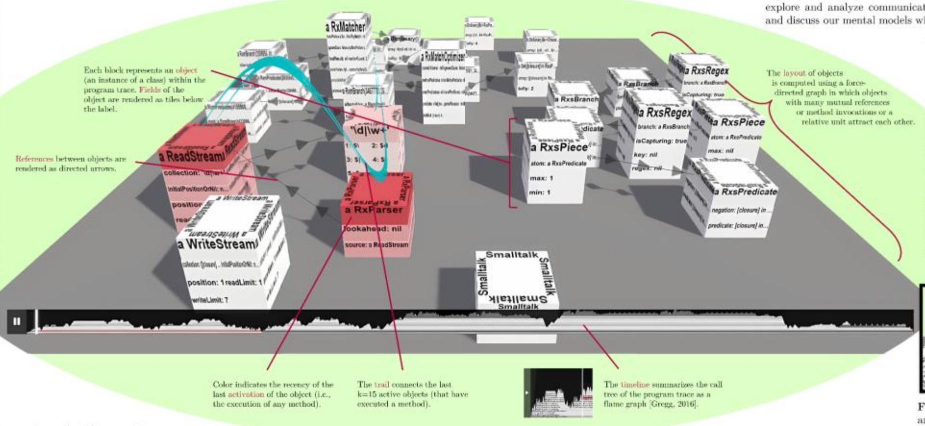
Background

Programmers often encounter familiar or unfamiliar software systems that they want to repair, modify, or extend. To understand these systems, they build up a mental model that links observable behavior to architectural and implementation units. For this, programmers traditionally start by studying the source code of systems. Because this approach is often hampered by abstractions and a lack of examples, *behavior-centric exploration* has become more popular in which programmers invoke a system with concrete inputs or test cases and explore its execution in a debugger to identify relevant units and their interactions by example.

Since traditional debuggers can only move forward along the execution time of a program, *omniscient debuggers* (or *time-travel debuggers*) allow for exploration of a recorded *program trace* in a nonlinear fashion [Pothier, 2009] (fig. 1). However, the displays of omniscient debuggers are too fine-grained and text-heavy to provide an overview of large program traces with multiple subsystems and many interacting objects, posing a need for inter-active visualization techniques that can be used to explore systems' behavior at large.

Approach

We record a trace of an object-oriented program and display it in a novel interactive *animated 2.5D object map* visualization (fig. 2). The visualization comprises an object map and a timeline. The object map displays each object as a cuboid block with its name and fields and arranges object blocks based on their classes and mutual references and interactions. As the animation plays, colors and a curved trail indicate the activation of objects that execute methods, and users can track the control flow throughout the object graph. The timeline provides an overview of the execution time and the call tree. Through the 3D view, more details can be added to the visualization and users can take different perspectives on the object graph.



Each block represents an object (an instance of a class) within the program trace. Fields of the object are rendered as tiles below the label.

References between objects are rendered as directed arrows.

The layout of objects is computed using a force-directed graph in which objects with many mutual references or method invocations or a relative unit attract each other.

Color indicates the recency of the last activation of the object (i.e., the execution of any method).

The trail connects the last k=15 active objects (that have executed a method).

The timeline summarizes the call tree of the program trace as a flame graph [Gregg, 2016].

Figure 2: An animated object map in our TRACE4D prototype showing a program trace for the parsing of a regular expression in the Squeak/Smalltalk programming environment.

Evaluation

We used our TRACE4D prototype to explore six different program traces from four different domains in the Squeak/Smalltalk programming system (tab. 1, fig. 3). Animated 2.5D object maps told us the most about program traces with a reasonably small number of relevant subsystems and objects as well as carefully designed systems that emphasize high class cohesion and extensive communication between related objects.

For suitable program traces, we could discover characteristic regions of the object graph (e.g., the input, the AST, and the NFA for the regular expression use case in fig. 1) as well as significant segments of program behavior (e.g., the parsing, compilation, and optimization stages). Based on these insights, we were able to develop and refine our mental model of the explored systems' functioning and link it to specific 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.

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	○	+	+	+	+

Table 1: Ratings of our experience with animated 2.5D object maps for program comprehension. We gained the most insights from smaller program traces that thoroughly model behavior through communication between objects and avoid many similar objects.

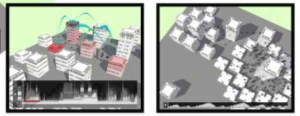


Figure 3: Animated object maps of an HTML parser (left) and a backtracking regular expression matcher (right).

References

Gregg, B. (2016). The flame graph. *Communications of the ACM*, 59(6):48–57.

Krause, A., Hansen, M., and Hasselbring, W. (2021). Live visualization of dynamic software cities with heat map overlays. In *Proc. VISSOFT*, pages 125–129. IEEE.

Limberger, D., Scheibel, W., Döllner, J., and Trapp, M. (2022). Visual variables and configuration of software maps. *Springer Journal of Visualization*, 26(1):249–274.

Pothier, G. and Tanter, É. (2009). Back to the future: Omniscient debugging. *IEEE Software*, 26(6):78–85.

Thiede, C., Taeumel, M., and Hirschfeld, R. (2023). Time-awareness in object exploration tools: Toward in situ omniscient debugging. In *Proc. SIGPLAN Onward!*, pp. 89–102. ACM.

Wettel, R. and Lanza, M. (2007). Visualizing software systems as cities. In *Proc. VISSOFT*, pages 92–99. IEEE.

Christoph Thiede
christoph.thiede@student.hpi.de
github.com/LinqLover

Willy Scheibel
willy.scheibel@hpi.de
hpi3d.de/people/current/scheibel.html

Jürgen Döllner
doellner@uni-potsdam.de
www.hpi3d.de

https://linqlover.github.io/trace4d/

Computer Graphics Systems Group
Hasso Plattner Institute
Prof.-Dr.-Helmert-Str. 2–3
D-14482 Potsdam, Germany

Universität Potsdam

HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

References

- Atzberger, D., Cech, T., Scheibel, W., Limberger, D., and Döllner, J. (2023). Visualization of source code similarity using 2.5D semantic software maps. In VISIGRAPP2021: Computer Vision, Imaging and Computer Graphics Theory and Applications, pages 162–182. Springer.
- Balzer, M., Deussen, O., and Lewerentz, C. (2005). Voronoi treemaps for the visualization of software metrics. InProc. SoftVis, pages 165–172. ACM.
- Boothe, P. and Badame, S. (2011). Animation of object-oriented program execution. In Proc. Bridges 2011:Mathematics, Music, Art, Architecture, Culture, pages 585–588. Tessellations Publishing.
- Brown, M. H. and Sedgewick, R. (1984). A system for algorithm animation. In Proc. SIGGRAPH, pages 177–186. ACM.
- Cheng, Y.-P., Chen, J.-F., Chiu, M.-C., Lai, N.-W., and Tseng, C.-C. (2008). XDIVA: A debugging visualization system with composable visualization metaphors. In Proc. SIGPLAN OOPSLA, pages 807–810. ACM.
- Chis, A., Gîrba, T., and Nierstrasz, O. (2014). The moldable debugger: A framework for developing domain-specific debuggers. In SLE 2014: Software Language Engineering, pages 102–121. Springer.
- Cornelissen, B., Zaidman, A., van Deursen, A., and van Rompaey, B. (2009). Trace visualization for program comprehension: A controlled experiment. In Proc. ICPC, pages 100–109. IEEE.
- Dashuber, V. and Philippsen, M. (2022). Trace visualization within the Software City metaphor: Controlled experiments on program comprehension. Elsevier In-formation and Software Technology, 150:55–64.
- Dugerdil, P. and Alam, S. (2008). Execution trace visualization in a 3D space. In Proc. ITNG, pages 38–43.IEEE.Fittkau,
- F., Waller, J., Wulf, C., and Hasselbring, W. (2013). Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In Proc. VISSOFT, pages 18:1–4. IEEE.
- Gestwicki, P. and Jayaraman, B. (2005). Methodology and architecture of JIVE. In Proc. SoftVis, pages 95–104. ACM.
- Gregg, B. (2016). The flame graph. Communications of the ACM, 59(6):48–57.
- Hamou-Lhadj, A. and Lethbridge, T. C. (2004). A survey of trace exploration tools and techniques. In Proc. CASCON, pages 42–55. IBM Press.
- Hamou-Lhadj, A. and Lethbridge, T. C. (2006). Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In Proc. ICPC, pages 181–190. IEEE.
- Harrower, M. and Brewer, C. A. (2003). ColorBrewer.org: An online tool for selecting colour schemes for maps. The Cartographic Journal, 40(1):27–37.
- Jerding, D. F. and Stasko, J. T. (1998). The Information Mural: a technique for displaying and navigating large information spaces. IEEE TVCG, 4(3):257–271.
- Ko, A. J. and Myers, B. A. (2008). Debugging reinvented: Asking and answering why and why not questions about program behavior. In Proc. ICSE, pages 301–310. ACM.
- Krause, A., Hansen, M., and Hasselbring, W. (2021). Live visualization of dynamic software cities with heat map overlays. In Proc. VISSOFT, pages 125–129. IEEE.
- Kruskal, J. B. and Landwehr, J. M. (1983). Icicle plots: Better displays for hierarchical clustering. Taylor & Francis The American Statistician, 37(2):162–168.
- Lange, D. B. and Nakamura, Y. (1997). Object-oriented program tracing and visualization. IEEE Computer,30(5):63–70.Langelier, G., Sahraoui, H., and Poulin, P. (2008). Exploring the evolution of software quality with animated visualization. In Proc. VLHCC, pages 13–20. IEEE.
- Lemieux, F. and Salois, M. (2006). Visualization techniques for program comprehension – a literature review. InProc. SoMeT, pages 22–47. IOS Press.
- Lienhard, A., Ducasse, S., and Gîrba, T. (2009). Takingan object-centric view on dynamic information with object flow analysis. Elsevier Computer Languages, Systems & Structures, pages 63–79.
- Limberger, D., Scheibel, W., Döllner, J., and Trapp, M. (2022). Visual variables and configuration of software maps. Springer Journal of Visualization, 26(1):249–274.
- Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M. (2004). Visualizing programs with Jeliot 3. In Proc. AVI, pages 373–376. ACM.
- Perscheid, M., Haupt, M., Hirschfeld, R., and Masuhara, H. (2012). Test-driven fault navigation for debugging reproducible failures. J-STAGE Information and Media Technologies, 7(4):1377–1400.
- Pothier, G. and Tanter, É. (2009). Back to the future: Omniscient debugging. IEEE Software, 26(6):78–85.
- Reiss, S. P. (2007). Visual representations of executing programs. Elsevier Journal of Visual Languages & Computing, 18(2):126–148.
- Scheibel, W., Limberger, D., and Döllner, J. (2020a). Survey of treemap layout algorithms. In Proc. VINCI, pages1:1–9. ACM.
- Scheibel, W., Trapp, M., Limberger, D., and Döllner, J.(2020b). A taxonomy of treemap visualization techniques. In Proc. IVAPP, pages 273–280. INSTICC, SciTePress.
- Scheibel, W., Weyand, C., and Döllner, J. (2018). EvoCells – a treemap layout algorithm for evolving tree data. In Proc. IVAPP, pages 273–280. SciTePress.
- Sorva, J., Karavirta, V., and Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. ACM Transactions on Computing Education, 13(4):1–64.
- Thiede, C. and Rein, P. (2023). Squeak by example. Lulu, 6.0 edition.
- Thiede, C., Taeumel, M., and Hirschfeld, R. (2023a). Object-centric time-travel debugging: Exploring traces of objects. In Proc. <Programming>, pages 54–60. ACM.
- Thiede, C., Taeumel, M., and Hirschfeld, R. (2023b). Time-awareness in object exploration tools: Toward in situ omniscient debugging. In Proc. SIGPLAN Onward!, pages 89–102. ACM.
- von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. IEEE Computer, 28(8):44–55.Walker, R. J., Murphy, G. C., Freeman-Benson, B., Wright, D., Swanson, D., and Isaak, J. (1998). Visualizing dynamic software system information through high-level models. ACM SIGPLAN Notices, 33(10):271–283.
- Waller, J., Wulf, C., Fittkau, F., Döhring, P., and Hasselbring, W. (2013). SynchroVis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency. In Proc. VISSOFT, pages 2:1–4. IEEE.
- Weninger, M., Makor, L., and Mössenböck, H. (2020). Memory Cities: Visualizing heap memory evolution using the software city metaphor. In Proc. VISSOFT, pages 110–121. IEEE.
- Wettel, R. and Lanza, M. (2007). Visualizing software systems as cities. In Proc. VISSOFT, pages 92–99. IEEE.