Towards Guided Omniscient Debugging in Education Using Pedagogical Execution Traces (PETs)

Markus Weninger
markus.weninger@jku.at
Institute for System Software
Johannes Kepler University Linz
Linz, Austria

Abstract

Educators frequently use trace-based debuggers for live classroom demonstrations. Yet, if a student's attention drops during class, they have to fall back to watching recordings (providing a passive, non-interactive experience) or replaying the debugging session at home (lacking the instructor's pedagogical context and verbal explanations). We introduce *Pedagogical Execution Traces (PETs)*, a concept that enriches execution traces with explanations, highlights and interactive questions. In this work-in-progress idea paper, we present the conceptual foundation of PETs as interactive learning artifacts, showing their applicability within JavaWiz, an educational trace-based graphical debugger. We explore PET authoring design goals and outline ongoing work regarding collaborative debugging scenarios and leveraging Large Language Models (LLMs) for trace annotation.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; Dynamic analysis; • **Applied computing → Interactive learning environments**; • **Social and professional topics** → *CS1*; • **Human-centered computing** → *Human computer interaction (HCI)*.

Keywords: Omniscient Debugging, Execution Traces, Software Engineering Education, Program Comprehension, Interactive Learning Artifacts, Educational Visualization

ACM Reference Format:
Markus Weninger. 2026. Towards Guided Omniscient Debugging in Education Using Pedagogical Execution Traces (PETs). In *Proceedings of Fourth Workshop on Future Debugging Techniques (DEBT'26).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DEBT'26, Brussels, Belgium
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Novices often struggle to form accurate mental models of program behavior and OOP concepts [8, 10, 24], a prerequisite for effective debugging. Live demonstrations [21], video recordings [14, 16], and time-traveling debuggers [28] each offer advantages on their own, yet they force educators to compromise between interactivity and pedagogical context. We address this gap with *Pedagogical Execution Traces (PETs)*.

We ground our work in the proven methodology of trace-based teaching [8] and implement our concepts within JavaWiz [28]. JavaWiz is an educational graphical time-travel debugger which is based on execution traces. An execution trace is a sequence of recorded events or program states that enable us to reconstruct the stack, heap, and statics fields (besides other trace-specific properties) at arbitrary points in time. In JavaWiz, these traces power various visualizations and enable users to step backward and forward through execution history. Currently, educators can save raw traces to disk and distribute them for students to replay at home.

PETs transform raw execution traces into guided learning artifacts by adding lightweight annotations, such as explanations, visual highlights, and interactive questions. Because these annotations are anchored to specific states and entities (e.g., a heap object or variable), they effectively clarify programming concepts. As a result, students can benefit from an instructor's insights without requiring live guidance.

In this work-in-progress idea paper, we outline the core design goals driving our development of PETs: student interaction possibilities, low-friction authoring, and a robust, view-agnostic data model, as well as future work.

2 Motivation and Design Goals

Based on experience with JavaWiz, to effectively preserve pedagogical intent within a trace, we identified three main design goals (DGs) for developing the PET concept:

DG1: Annotations Must Support Novice Comprehension. To mirror live in-class guidance, PETs must support:

- contextual explanations where misconceptions typically arise (e.g., object aliasing or off-by-one errors).
- non-textual highlights to direct attention
- formative assessments (e.g., quizzes) that test learning outcomes and optionally gate the next execution step.

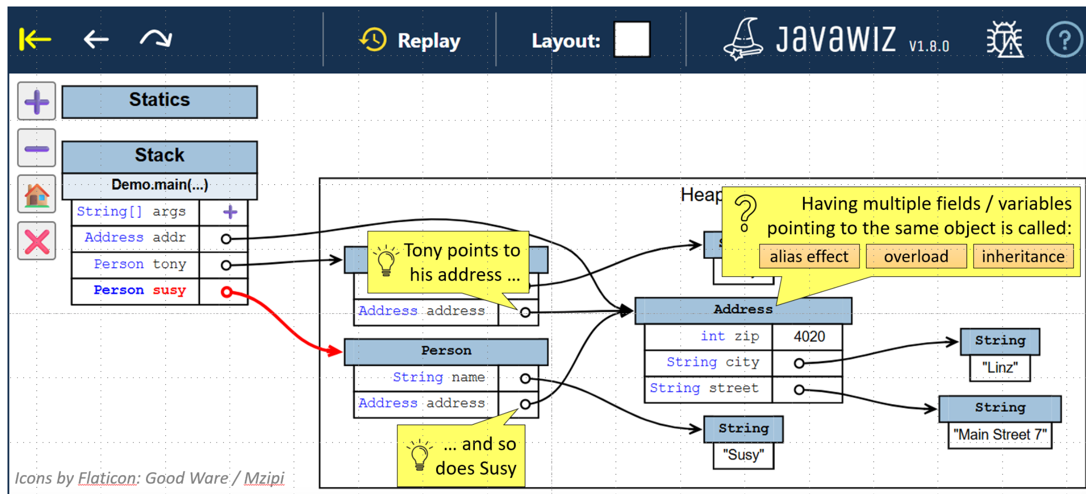

Figure 1. Mockup of JavaWiz's annotated *Memory View*. The PET comments in line 6-13 of Listing 1 translate to speech bubbles attached to `field tony.address` and `field susy.address` as well as a question linked to the heap `object tony.address`.

Listing 1. PETs are designed via "pedagogy-as-code", i.e., via comments that reflect the instructor's educational intent.

```
1  /* @PET Say in Controls at button StepInto: You can use
2          Step Into to inspect the constructor */
3  Address a = new Address(4020, "Linz", "Main Street 7");
4  Person tony = new Person("Tony", a);
5  Person susy = new Person("Susy", a);
6  /* @PET Say in MemoryView at field tony.address:
7          Tony points to his address... */
8  /* @PET Say in MemoryView at field susy.address:
9          ... and so does Susy */
10 /* @PET AskSingle in MemoryView at object tony.address:
11         Having multiple fields/variables pointing to
12         the same object is called
13         (*alias effect*|overload|inheritance) */
```

DG2: Minimize Instructor Effort for Producing Maintainable PETs. Manual UI-based annotation is tedious and brittle. We strive for a low-friction approach by embedding the instructor's educational intent as comments *within* the source code – we call this "*pedagogy-as-code*". These comments compile into annotations and, crucially, survive minor code edits without requiring instructors to re-annotate.

DG3: Binding Annotations to Program Entities. Annotations must be tied to semantic program entities (e.g., objects, fields, or variables) rather than UI coordinates. Annotations must be view-agnostic (rendering consistently across different visualizations) and packaged within the trace.

3 Idea: Pedagogical Execution Traces (PETs)

We propose *Pedagogical Execution Traces (PETs)*, traces enriched with interactive *Pedagogical Annotations (PAs)* attached to individual states (or, in future work, multiple states or state transitions). We outline the conceptual foundation of PETs, structured along our three core design goals.

3.1 Interaction Possibilities (DG1)

To effectively mirror the guidance an instructor provides during a live debugging session, we distilled the required interactions into three fundamental annotation types:

- **Say (Explanations):** Contextual "speech bubbles" embedded directly within the visualizations. These bubbles are attached to specific entities (e.g., a newly allocated object in the heap or a specific method frame on the stack) at the exact moment they become relevant.
- **Highlights (Focus Points):** Visual highlights (e.g., colored borders or dimming the surrounding view) to direct the learner's attention to a specific graphical element. This is crucial for guiding the eye during complex state changes (e.g., linked list operations)
- **Ask (Questions):** Assessments embedded within the trace. These can take the form of single-choice, multiple-choice, or short free-text questions. Crucially, "Ask" annotations can act as *execution gates*, requiring the student to predict the next state (e.g., "Which variable will change in the next step?") before allowing them to continue stepping through the trace.

3.2 Authoring Tools (DG2)

A major adoption barrier for educational tools is authoring friction. Initially, we considered a graphical "authoring mode" within JavaWiz, where instructors could manually place annotations on the screen after recording a trace. However, coordinate-based placements break across different screen sizes, and even minor changes to the source code would require the instructor to re-record and re-annotate the entire session. Instead, we propose a "pedagogy-as-code" approach using structured source code comments that are automatically compiled into the execution trace. Comments also allow the pedagogy to be version-controlled alongside the code.

The example in Listing 1 could be visualized as Figure 1 and demonstrates how an educator can annotate code to demonstrate the concept of object aliasing. While the exact syntax is still work-in-progress, we target a declarative *What-Where-Target-When* grammar with optional parts as follows:

```
/* @PET <Action> [in <View>] [at <Target>]
        [when <Condition>]: <Payload> */
```

Action is either "Say", "Highlight", or "Ask". *View* restricts the annotation to a specific visualization (e.g. the Memory View). *Target* defines at which entity the annotation should be anchored (see next section), and *Condition* (e.g., `when i == 0`) will allow instructors to annotate loops and recursive calls in more detail. This structure is designed to be highly readable for humans while keeping distraction low.

Regarding workflow, the annotated source code is an instructor authoring tool. When exported for students, `@PET` comments are stripped from the embedded code and stored purely as structural metadata within the trace states.

3.3 Identifying Program Entities (DG3)

PAs are anchored to semantic targets to remain stable even if the instructor adds or removes lines of code. This system also ensures that annotations are view-agnostic and layout-independent. The frontend simply queries the graphical node associated with the given target and renders the annotation accordingly. Crucially, as presented in Figure 1, we distinguish between variables (e.g., `field tony.address`) and objects (e.g., `object tony.address`). Table 1 lists entities that our initial version of PETs for JavaWiz will support.

Table 1. The different targets for Pedagogical Annotations.

Group	***Kind***	***Example***
Variables	Local	*at local x*
	Static	*at static Clazz.MAX*
	Array Element	*at element arr[4]*
	Fields	*at field tony.address*
Objects	Object	*at object tony.address*
Control Flow	Class	*at class Clazz*
	Method	*at method Clazz.foo*
	Line	*at line Clazz:8*
JavaWiz-Specific	Buttons	*at button StepInto*
	...	...

4 State-of-the-art and Related Work

Omniscient debuggers / Capture-and-replay debuggers capture full execution histories, while *time-traveling debuggers* provide retrospective backward/forward navigation [12, 17–20, 23, 26, 27]. These approaches rely on dynamic analysis over execution traces [4, 7, 13, 22] (see Cornelissen et al. [5] for a comprehensive survey). PETs leverage the same foundations but introduce a layer that encodes pedagogical intent directly into the trace.

Educational visualizers such as Jeliot [1–3, 15], jGRASP [6], Python/Java Tutor [9], Localizer [11], and various heap visualization tools [29, 30] help novices build mental models of runtime concepts (see Sorva et al. [25] for a comprehensive survey). However, while these systems visualize executions, they do not package embedded pedagogy. Building on JavaWiz [28], PETs differentiate themselves by binding explanations and questions to specific semantic entities via an easy-to-maintain "pedagogy-as-code" authoring model.

5 Ongoing and Future Work

PETs open several avenues for future research, ranging from pedagogical evaluations to advanced AI integrations.

5.1 Student Interaction Styles

To measure the pedagogical impact of PETs, we plan to conduct user studies comparing cognitive load, engagement, and retention across three scenarios: live instructor presentations, passive video recordings, and active PET stepping. We hypothesize that PETs' self-paced nature and "Ask" execution gates will foster active learning and increase retention.

5.2 Collaborative Educational Scenarios

PETs enable "Debug With Me" sessions: students follow an instructor's debugging session live on their own devices and place annotations to answer instructor prompts (e.g., predicting memory changes). The instructor can later review class-annotated PETs to gauge student engagement / comprehension. Further, having students author PET comments themselves as homework (or collaboratively during live coding) turns the annotation process itself into an active pedagogical exercise in predicting program behavior [8].

5.3 AI-Assisted and Autonomous Debugging

To further reduce authoring friction, we investigate LLMs for automated PET generation. We currently conduct a study on optimizing LLM inputs (source code vs. raw trace vs. images) to maximize explanation quality while minimizing latency. Additionally, we compare student ratings of AI-generated versus human-authored PETs. We further envision stronger LLM integration into JavaWiz via the Model Context Protocol (MCP). Rather than statically generating PETs, an LLM could then autonomously run JavaWiz, dynamically step through code, open relevant visualizations, and inject real-time annotations based on perceived student struggles, effectively acting as a personalized, interactive tutor.

6 Conclusion

Traditional debuggers are powerful diagnostic tools, but they lack the pedagogical context required for effective software engineering education. While live demonstrations provide this context, they are ephemeral; while video recordings preserve it, they sacrifice interactivity.

In this work-in-progress idea paper, we introduced the concept of *Pedagogical Execution Traces (PETs)* to bridge this gap. Extending the trace-based debugging capabilities of JavaWiz, PETs transform execution traces into reusable, self-paced learning artifacts. We outlined a "pedagogy-as-code" authoring approach that allows instructors to express explanations, highlights, and questions via source code comments.

Looking forward, the integration of LLMs for automated annotation and the development of collaborative, in-class debugging scenarios represent exciting frontiers. Ultimately, we envision that concepts such as our PETs can help debuggers to evolve beyond mere diagnostic utilities into active, intelligent partners in the learning process.

References

[1] Mordechai Ben-Ari, Roman Bednarik, Ronit Ben-Bassat Levy, Gil Ebel, Andrés Moreno, Niko Myller, and Erkki Sutinen. 2011. A decade of research and development on program animation: The Jeliot experience. *J. Vis. Lang. Comp.* 22, 5 (2011), 375–384. doi:10.1016/J.JVLC.2011.04.004

[2] Mordechai Ben-Ari, Niko Myller, Erkki Sutinen, and Jorma Tarhio. 2001. Perspectives on Program Animation with Jeliot. In *Software Visualization, Int'l Seminar Dagstuhl Castle (Lecture Notes in Computer Science, Vol. 2269)*. 31–45. doi:10.1007/3-540-45875-1_3

[3] Sanja Maravic Cisar, Robert Pinter, and Dragica Radosav. 2011. Effectiveness of Program Visualization in Learning Java: a Case Study with Jeliot 3. *Int. J. Comput. Commun. Control* 6, 4 (2011), 668–680. doi:10.15837/IJCCC.2011.4.2094

[4] 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. *J. Syst. Softw.* 81, 12 (2008), 2252–2268. doi:10.1016/j.jss.2008.02.068

[5] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. 2009. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Trans. Software Eng.* 35, 5 (2009), 684–702. doi:10.1109/TSE.2009.28

[6] J.H. Cross, Dean Hendrix, and Larry Barowski. 2011. Combining Dynamic Program Viewing and Testing in Early Computing Courses. In *Int'l Computer Software and Applications Conf. (COMPSAC)*. 184 – 192. doi:10.1109/COMPSAC.2011.31

[7] Tobias Herber and Markus Weninger. 2025. Trace-Based Bytecode Interpreter Visualization for Compiler Construction Education. In *IEEE Working Conf. on Software Visualization (VISSOFT)*. IEEE, 13–24. doi:10.1109/VISSOFT67405.2025.00010

[8] Matthew Hertz and Maria Jump. 2013. Trace-based teaching in early programming courses. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 561–566. doi:10.1145/2445196.2445364

[9] Java Tutor. 2026. Java Tutor. https://pythontutor.com/java.html.

[10] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education, SIGCSE 2010, Milwaukee, Wisconsin, USA, March 10-13, 2010*. ACM, 107–111. doi:10.1145/1734263.1734299

[11] Shehroz Khan, Gaadha Sudheerbabu, Dragos Truscan, and Tanwir Ahmad. 2024. Localizer: A Visual Debugging Assistant for Python Programs. In *ACM Int'l Workshop on Future Debugging (DEBPT)*. ACM, 34–35. doi:10.1145/3678720.3685321

[12] M. A. Klimushenkova and P. M. Dovgalyuk. 2017. Improving the Performance of Reverse Debugging. *Program. Comput. Softw.* 43, 1 (2017), 60–66. doi:10.1134/S0361768817010042

[13] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Int'l Conf. on Principles and Practices of Programming on the Java Platform (PPPJ)*. ACM, 4:1–4:11. doi:10.1145/2972206.2972220

[14] Jill R. D. MacKay. 2019. Show and 'tool': How lecture recording transforms staff and student perspectives on lectures in higher education. *Comput. Educ.* 140 (2019). doi:10.1016/J.COMPEDU.2019.05.019

[15] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. 2004. Visualizing programs with Jeliot 3. In *Working Conf. on Advanced Visual Interfaces (AVI) (AVI '04)*. 373–376. doi:10.1145/989863.989928

[16] Frances V. O'Callaghan, David L. Neumann, Liz Jones, and Peter A. Creed. 2017. The use of lecture recordings in higher education: A review of institutional, student, and lecturer issues. *Educ. Inf. Technol.* 22, 1 (2017), 399–415. doi:10.1007/S10639-015-9451-Z

[17] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *USENIX Ann. Techn. Conf.* 377–389. https://www.usenix.org/Conf./atc17/technical-sessions/presentation/ocallahan

[18] Guillaume Pothier and Éric Tanter. 2008. Extending Omniscient Debugging to Support Aspect-Oriented Programming. In *Symposium on Applied Computing (SAC)*. , 266–270. doi:10.1145/1363686.1363753

[19] Guillaume Pothier and Éric Tanter. 2009. Back to the Future: Omniscient Debugging. *Softw.* 26, 6 (2009), 78–85. doi:10.1109/MS.2009.169

[20] Guillaume Pothier, Éric Tanter, and José M. Piquer. 2007. Scalable Omniscient Debugging. In *Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. , 535–552. doi:10.1145/1297027.1297067

[21] Adalbert Gerald Soosai Raj, Jignesh M. Patel, Richard Halverson, and Erica Rosenfeld Halverson. 2018. Role of Live-coding in Learning Introductory Programming. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research, Koli, Finland, November 22-25, 2018*. ACM, 13:1–13:8. doi:10.1145/3279720.3279725

[22] Steven P. Reiss and Manos Renieris. 2001. Encoding Program Executions. In *Int'l Conf. on Software Engineering (ICSE)*. IEEE Computer Society, 221–230. doi:10.1109/ICSE.2001.919096

[23] Kazumasa Shimari, Takashi Ishio, Tetsuya Kanda, and Katsuro Inoue. 2024. Evaluating the Effectiveness of Size-Limited Execution Trace with Near-omniscient Debugging. *Sci. Comput. Program.* 236 (2024), 103117. doi:10.1016/J.SCICO.2024.103117

[24] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 2 (2013), 8:1–8:31. doi:10.1145/2483710.2483713

[25] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 4 (2013), 15:1–15:64. doi:10.1145/2490822

[26] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. Time-Awareness in Object Exploration Tools: Toward In Situ Omniscient Debugging. In *SIGPLAN Onward!* , 89–102. doi:10.1145/3622758.3622892

[27] Marnix van 't Riet, Haihan Yin, and Christoph Bockisch. 2013. The Potential of Omniscient Debugging for Aspect-Oriented Programming Languages. In *Workshop on Comprehension of Complex Systems (CoCoS)*. , 13–16. doi:10.1145/2451592.2451597

[28] Markus Weninger, Simon Grünbacher, and Herbert Prähofer. 2025. JavaWiz: A Trace-Based Graphical Debugger for Software Development Education. In *33rd IEEE/ACM Int'l. Conf. on Program Comprehension (ICPC@ICSE 2025)*. IEEE, 1–12. doi:10.1109/ICPC66645.2025.00023

[29] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *Working Conf. on Software Visualization (VISSOFT)*. 110–121. doi:10.1109/VISSOFT51673.2020.00017

[30] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations. In *Smart Tools and Apps in Computer Graphics (STAG)*. 63–75. doi:10.2312/STAG.20201241