

LIVE:  
Empirical Studies on  
the Effects of Liveness on Programming  
Final Report

*DFG reference number*

HI 1385/3-1

*Project number*

449591262

*Project title*

LIVE: Empirical Studies on the Effects of Liveness on Programming

*Name of the applicant*

Prof. Dr. Robert Hirschfeld

*Official address*

Hasso Plattner Institute for Digital Engineering gGmbH

Prof.-Dr.-Helmert-Str. 2-3

14482 Potsdam, Germany

*Reporting period*

April 2021 - March 2024

## 2 Summary

### 2.1 Summary

Liveness in programming tools is the impression of changing a program while it is running. Various tools support liveness, including commercial programming systems, such as MS Excel and Jupyter Notebooks. Tool designers assume that liveness improves the programming experience, but this assumption has insufficient and inconclusive empirical backing. This lack of evidence might lead to the promotion of liveness in unsuitable settings and the neglect of important settings, which would waste design and implementation efforts.

In this project, we investigated the effects of live tools on debugging. In two controlled experiments we studied the influence of task complexity and delayed interactions on the effects of live tools. Compared to previous experiments on liveness, the participants in our experiments had considerable experience with live tools.

In our first experiment we tested whether the influence of live tools on debugging time differs for simple and complex tasks. We found that live tools significantly shorten the time needed to debug defects. At the same time, we could not confirm our main hypothesis that task complexity moderates this effect. However, our results indicate that task complexity indeed influences the effect, but less than suggested by the pilot. For programming tool researchers and designers, our results show that programmers can benefit from live tools, but that they need to consider task complexity and participants' experience with liveness when preparing studies or building tools.

With our second experiment, we aimed to better understand the first experiment's observations. Based on Information Foraging Theory, we assumed that live tools reduce the perceived cost of obtaining dynamic information so that programmers consult it more often when helpful. Therefore, we tested whether programmers use live tools less frequently if access to them is delayed. The experiment did not yield sufficiently enough results for a thorough analysis, but the collected data shows no clear decline in live tool usage. Yet, an ongoing post hoc analysis using edit-run cycles suggests that participants' workflows changed.

During the first experiment, we found that it is a great challenge to operationalize the complexity of maintenance tasks in programming tool studies. Thus, we conducted a survey to curate a collection of factors from related studies that can help shape the complexity of such tasks. With this collection, researchers can deliberately decide on the complexity level for their studies' tasks.

This project also resulted in a novel concept for teaching debugging through contests and improved setups for related studies on liveness conducted in our group.

## 2.2 Zusammenfassung

Liveness in Programmierwerkzeugen ist der Eindruck, ein Programm zu verändern, während es läuft. Viele Werkzeuge einschließlich kommerzieller Programmiersysteme wie MS Excel und Jupyter Notebooks unterstützen Liveness. Während Werkzeugentwickelnde annehmen, dass Liveness die Programmiererfahrung verbessert, ist diese Annahme nur unzureichend empirisch belegt. Dieses fehlende Wissen kann dazu führen, dass Liveness nicht bestmöglich eingesetzt wird und so unnötiger Entwurfs- und Implementierungsaufwand entsteht.

In diesem Projekt haben wir untersucht, wie Live-Werkzeuge das Debugging beeinflussen. In zwei kontrollierten Experimenten untersuchten wir die Auswirkungen von Aufgabenkomplexität und verzögerten Interaktionen auf den Einfluss von Live-Werkzeugen. Im Gegensatz zu früheren Experimenten hatten unsere Teilnehmenden substanzielle Erfahrung mit diesen.

Das erste Experiment testete, ob der Einfluss von Live-Werkzeugen auf die Debugging-Dauer bei einfachen und komplexen Aufgaben unterschiedlich ist. Das Experiment hat gezeigt, dass Live-Werkzeuge die Debugging-Dauer signifikant verkürzen. Die Haupthypothese, dass die Aufgabenkomplexität den Effekt beeinflusst, hat sich nicht bestätigt. Dennoch weisen die Ergebnisse darauf hin, dass die Aufgabenkomplexität die Auswirkungen beeinflusst, allerdings weniger als der Pilot vermuten ließ. Für Forschende und Entwickelnde zeigen die Ergebnisse, dass Programmierende von Live-Werkzeugen profitieren können, aber auch, dass die Aufgabenkomplexität und die Erfahrung der Teilnehmenden mit Liveness beim Planen von Studien oder der Entwicklung von Werkzeugen berücksichtigt werden sollten.

Mit dem zweiten Experiment wollten wir die Ergebnisse des ersten Experiments besser verstehen. Nach der Information-Foraging-Theorie gingen wir davon aus, dass Live-Werkzeuge die wahrgenommenen Kosten für den Zugriff auf dynamische Informationen verringern, so dass Programmierende häufiger auf sie zugreifen, wenn sie hilfreich sind. Daher haben wir getestet, ob Programmierende Live-Werkzeuge seltener verwenden, wenn der Zugriff auf sie durch eine künstliche Verzögerung erschwert wird. Die erhobenen Daten reichen nicht für eine volle Analyse, zeigen aber auch keinen deutlichen Rückgang der Nutzung. Eine laufende post hoc-Analyse anhand von Edit-Run-Zyklen deutet jedoch darauf hin, dass sich die Arbeitsabläufe der Teilnehmenden verändert haben.

Um andere Forschende darin zu unterstützen, die Komplexität der Aufgaben in ihren Studien zu verstehen und zu formen, haben wir eine Sammlung von Faktoren erstellt und publiziert, die dabei hilft, die Komplexität von Programmierungsaufgaben zu beeinflussen. Diese Sammlung erlaubt es Forschenden, sich bewusst für ein Komplexitätsniveau ihrer Aufgaben zu entscheiden.

Außerdem führte unser Projekt zu einem neuartigen Konzept, Debugging durch Wettbewerbe zu lehren, und zu verbesserten Anordnungen für weitere Liveness-Studien in unserer Gruppe.

### 3 Progress Report

This project's objective was to learn more about the effects of liveness. Many approaches to introduce liveness into programming environments argue that liveness is beneficial in multiple ways [1, 2, 3], and studies find that generally, programmers appreciate liveness in their tools [4, 5]. Still, experiments on liveness so far have had inconclusive results, with most showing no improvement in programmer performance [5, 6, 7]. Based on additional insights from human-computer interaction research, we concluded that two major influences have been neglected in studies of liveness: *task complexity* and *participants' experience with liveness*.

We planned two experiments around the main hypothesis that task complexity moderates the impact of live programming tools for programmers with experience with live programming. We planned to investigate the impact of two aspects of liveness: live introspection tools and different feedback modes. We conducted two corresponding controlled experiments. The first experiment tested the moderation effect of task complexity on the influence of live introspection tools on debugging efficiency. Based on the experience with the first experiment, we adapted the hypothesis for the second experiment and tested the influence of delayed feedback on live tool usage when participants work on complex tasks.

Through the work on these experiments, the project yielded the following results (for details, see the following sections):

- The first experiment's results show a significant main effect, so live introspection tools significantly improved debugging efficiency. The moderation effect of task complexity was not significant, but the data showed a trend. With 37 valid results, the sample size was likely too small to show the moderation effect, especially as the effect size was smaller in the actual experiment run than in the pilot. The results were published in *The Programming Journal* [R1]<sup>1</sup>.
- We collected and curated a set of task complexity factors that can be used to control or evaluate the complexity of tasks used in studies on programming tools. Originally, we gathered a first set of task complexity factors to design the tasks for experiment 1. As the collection became useful in the design of other studies, we improved and published the collection by expanding the underlying survey. The results were presented at the ICPC conference [R2] and the PX workshop [R3].
- The second experiment did not yield sufficient data due to considerable recruiting challenges throughout the experiments. Also, the collected data shows that participants used live introspection tools even when response times were artificially delayed. However, first results from an ongoing, post hoc analysis based on the edit-run cycle model [8] show clear changes in the workflow. We are preparing a submission for ICPC 2025.

---

<sup>1</sup>References prefixed with *R* refer to publications of project results.

- Additionally, the project resulted in a novel concept for teaching debugging techniques called *competitive debugging*, which is analogous to *competitive programming* [R4]. Also the project lead to a multiplication of knowledge on liveness and empirical studies through collaborating with colleagues within the group and consulting on methodology of related projects, resulting in several publications including studies with programmers [R5, R6, R7, R8, R9, R10].

### 3.1 Experiment on Task Complexity and Live Tools

The main hypothesis for the first experiment was that “task complexity moderates the impact of tool support for introspection on debugging performance for programmers who are experienced with live programming tools.” The main results of the experiment were published in the journal “The Art, Science, and Engineering of Programming” [R1]. In the following, we recap the main points of the setup and the results of the experiment:

**Setup** We conducted the experiment as a within-subject experiment using a 2x2 factorial design. The main independent variables for this experiment were the availability of live tools and the complexity of the provided tasks.

For this experiment, we investigated exploratory-programming-style liveness, which is characterized by evolving a running system and strong tool support for live introspection of run-time state. In this study, we used the Squeak/Smalltalk environment [9], as it offers a large variety of live introspection tools for a general-purpose, object-oriented language and our prospective subjects had considerable experience in using it. In the control condition (*with*), all Squeak/Smalltalk live introspection tools were available, and the Smalltalk hot-swapping was active. In the experimental condition (*without*), access to these tools was restricted, resulting in a programming experience more similar to a basic code editor, a step-wise debugger, and a manual recompilation step.

The operationalization of the second variable, task complexity, is more involved, as the complexity of a task results from many properties of the tasks. To control task complexity, we created a collection of task complexity factors based on a general model of task complexity from industrial ergonomics research (see subsequent Section). Using this collection we designed tasks with two complexity levels, with tasks in one group (*simple*) being simpler in relation to tasks in the other group (*complex*). While we kept many properties similar between all tasks, the simple tasks all contained a fault of commission (for instance wrong message selector, wrong class name, mixed up conditional branch), while the complex tasks all contained a fault of omission (for instance missing message sends). Thus, in *simple* tasks, participants could spot the fault in the code, while for *complex* tasks, they had to understand

the program behavior to determine that a relevant operation was missing.

The main dependent variables were debugging efficiency and debugging effectiveness. We operationalized the debugging efficiency as the time to solve a task, from successfully reproducing the failure until participants submitted their patch. We operationalized debugging effectiveness as the number of correctly solved tasks. We also measured the usage frequency of live tools to ensure that any observed effects actually result from changed usage patterns.

The participants for this study were undergraduate students ( $N = 37$ ) experienced with liveness due to nine months of exposure to Squeak/Smalltalk in two lecture-accompanying projects. Participants worked on four to eight tasks in a four-hour session. Tasks were counterbalanced with a Latin square scheme.

We tested the task setups and the experiment procedures in eight test runs with graduate students. After ensuring that the process was robust, we conducted a pilot with four participants from the target population. Before conducting the main run of the experiment, we prepared a full write-up of the complete methodology and the statistical analysis.

**Results** We analyzed the results of the main run using a standard two-way repeated measures ANOVA.

The moderation effect of task complexity on debugging efficiency was not statistically significant in our results ( $F(1, 36) = 1.84, p = 0.18$ ). However, the data hinted at a moderation effect, and the number of participants was likely insufficient. The sample size estimation did not show this in advance as the effect in the pilot results (conducted with participants from the target group) was stronger than in the results from the full run.

At the same time, the results showed that the overall effect of live introspection tools on debugging efficiency was statistically significant ( $F(1, 36) = 4.54, p = 0.04$ ). The live introspection tools did reduce the time participants needed to solve the tasks.

Finally, there was no effect of live introspection tools on debugging effectiveness. There were almost no incorrect patches, which might have resulted from us priming participants to work until they were sure of their patch.

### **3.2 Collection of Task Complexity Factors**

To evaluate and control task complexity during the experiments, we collected aspects of tasks that contribute to task complexity from related studies on programming tools. To make the collection accessible for other researchers interested in programming tools, we organized it along dimensions of task complexity from an abstract model of task complexity and typical artifacts of studies on programming tools. We presented the refined collection

at the International Conference on Program Comprehension (ICPC) 2023 [R2]. The paper contains the full mapping of factors to publications of studies that used them to control task complexity. We also presented a first version of the collection at the Programming Experience Workshop (PX) 2022 to gather feedback from the community [R3].

The collection is the result of surveying 62 papers describing studies on programming tools that included maintenance tasks (corrective or perfective). In total, it features more than 75 factors. We organized them along the typical artifacts of maintenance tasks, which we identified as task description, system, infection chain or feature location, patch, tool environment, and overall considerations affecting the context. For each artifact, we further grouped factors according to general components of task complexity as identified by a general framework of task complexity from industrial ergonomics research [10]: input, goal/output, process, presentation, and time.

The collection allowed us to make deliberate choices when designing the tasks for our experiments, and we hope that other researchers will find it equally useful when preparing tasks for their studies.

### **3.3 Experiment on Delayed Interactions**

While the first experiment has shown that live tools improve debugging efficiency, the mechanism that makes programmers more efficient remains unclear. In particular, the question remains whether the introspection features themselves already make programmers more efficient or whether the tools also need to be live to be effective. Thus, we designed the second experiment to better understand the role of liveness in live introspection tools. The analysis of the results is still in process and we are preparing a submission of the results to the International Conference on Program Comprehension (ICPC) 2025.

We used Information Foraging Theory to reason about the relationship between liveness, introspection tools, and debugging efficiency. Information Foraging Theory (IFT) [11] posits that users optimize their information-searching behavior in a graph of information artifacts by optimizing the ratio between the information gained from an artifact and the effort required to obtain it. Before deciding what artifact to investigate next, users estimate the benefit and effort for getting artifacts and decide based on this estimated benefit-effort-ratio. IFT can also be applied to model and predict the information seeking behavior of programmers [11], including information seeking during debugging [12]. Based on the model of IFT, introspection tools can reduce the cost of accessing dynamic information. Adding liveness to these tools, can also reduce the perceived cost of accessing dynamic information, which should result in programmers using dynamic information more often. We hypothesized that using dynamic information more often in debugging shortens the time to debug a problem, as it allows programmers to trace the infection chain more closely than mentally tracing it from the source

code. Based on this reasoning, we adapted the hypothesis for this experiment to “feedback mode (normal, delayed) influences tool usage of introspection tools for programmers who are experienced with live programming tools working on complex tasks.”

The original hypothesis for this experiment was that task complexity would moderate the effects of feedback modes on debugging performance. As the data from the first experiment suggested that the impact of live tools is larger for complex tasks, we decided to focus on the effect of the feedback mode directly and only use complex tasks for this experiment.

**Setup** As with the first experiment, we conducted the experiment as a within-subject experiment. The independent variable for this experiment was the feedback mode of the live introspection tools. Again, we conducted the experiment in the Squeak/Smalltalk programming environment.

We distinguished between two feedback modes: *normal* and *delayed*. In the *normal* condition, programmers could use the live introspection tools directly without any modifications. In the *delayed* condition, programmers experienced a delay in some interactions with live introspection tools. The delay varied between eight and twelve seconds and was added to all interactions with live introspection tools that would provide the participants with new information. This involved the opening of any live introspection tool (object inspector and explorer, Halo, step-wise debugger, interactive code evaluation), as well as navigating in the tree view of the object explorer, as opening a sub-tree is equivalent to opening another object inspector. Selecting user interface elements and other navigation interactions were not delayed.

We kept the task complexity at the level of complex tasks from the first experiment, as the results suggested that we would observe a greater effect with complex tasks. We used a scenario comparable to the one in the first experiment. Again, tasks were counterbalanced using a Latin square scheme, and participants worked on six tasks during a four-hour session. Participants worked on the first half of the tasks in one feedback mode and on the second half in the other mode. To accommodate for the time participants would need to adapt to the changed feedback mode, there was a fifteen-minute break between the first and the second half. Further, the very first task and the first task after the break were both warm-up tasks and were excluded from the analysis.

The main dependent variable for this experiment was tool usage. We operationalized tool usage through three different metrics to prevent introducing bias: clicks, number of opened tools, and time spent using a tool. While these metrics depend on each other, they still offer three different perspectives, each with its own interpretations.

As the target population was not available anymore, we extended the population to include all

students who had nine months of exposure to Squeak/Smalltalk. To accommodate for long breaks between participants' Squeak/Smalltalk experience and the experiment, we added a mandatory two-hour training session for all participants on the days before the main run. Despite the much larger target population and persistent recruiting efforts, we had a very weak turnout for this experiment, with only twelve valid runs.

**Experiment Results** In addition to the weak turnout, we could also observe during the runs that participants kept using the live tools extensively in the delayed mode. This was in contrast to the pilot runs, during which participants clearly avoided the live tools in the delayed mode. As we suspected that the experiment setup might not be suitable and recruiting got increasingly difficult and less effective, we decided not to conduct further runs to not waste effort. An analysis of the interaction metrics showed that the impression from our direct observations was indeed correct, and the feedback mode did not influence the usage frequency of live tools.

**Preliminary Edit-Run Analysis** To better understand whether and how the delayed condition affected participants, we decided to conduct a detailed post hoc analysis of participants' behavior using the edit-run cycle model [8]. The edit-run cycle model was originally used to analyze how programmers alternate between working with the code (edit) and observing program behavior either through surface behavior or run-time state (run). To apply this analysis, we started manually coding the recordings of the participants using the code book of the original study and have so far processed half of the recordings (thus, the following results should be regarded as preliminary).

For the subsequent analysis, we retrieved the analysis infrastructure from the original edit-run cycle study and adapted it to our setting. The edit-run cycle analysis confirms that the usage duration and frequency of live introspection tools did not change consistently between feedback modes. However, it suggests that the feedback mode did influence participants' workflow. For one, the number of edit-run cycles is clearly lower in the delayed mode. Further, the edit-run cycles that occur in the delayed mode are longer, with almost no edit-run cycles shorter than a minute.

The most relevant result with regard to our initial hypothesis is that in the delayed condition, participants used surface behavior almost 30% of the time they inspected program behavior, while they otherwise only used it 10% of the time. This suggests that while participants did value dynamic information, they changed how they access it in the delayed condition.

While interesting, these results should be regarded as preliminary and unreliable.

### 3.4 Further Results

**Competitive Debugging** As a result of the feedback from participants who enjoyed the experiment setup for experiment 1, we devised a novel format for teaching debugging called *Competitive Debugging*. We presented the format and experiences with first debugging contests at Symposium for New Ideas, New Paradigms, and Reflections on Everything to do with Programming and Software (Onward!) 2022 [R4].

The format is motivated by the fact that debugging is a major part of modern software development and is also a skill that needs to be learned and trained for programmers to become good at it. We proposed Competitive Debugging as a way to promote debugging as a skill and to spread knowledge about debugging techniques and tools. In Competitive Debugging, participants find and repair prepared failures of a software system and are rated on various scales, such as the time they needed to repair the failure, the correctness of the patch, or the degree to which the underlying fault was repaired. We conducted three events to explore possible variations of debugging contests. We also used these test events to select a system and potential tasks for experiment 2.

**Multiplicator of Knowledge about Study Design and Liveness** Preparing and conducting the experiments for this project led to numerous further studies with programmers within the research group. The initial team working on the experiments continued to collaborate with other members of the group on study designs for other projects, leading to a multiplication of knowledge and skills on user studies and several publications involving user studies and experiments [R5, R6, R7]. The expertise gained from this project on the concepts and effects of liveness also has had further impact. We have collaborated with members of the group to advance approaches using liveness to make information more immediately available in programming environments, resulting in several publications [R8, R9, R10].

### 3.5 Bibliography

- [1] Rein, Ramson, Lincke, Hirschfeld, and Pape. “Exploratory and Live, Programming and Coding - A Literature Study Comparing Perspectives on Liveness”. In: *The Art, Science, and Engineering of Programming* 3.1 (July 2019). DOI: 10.22152/programming-journal.org/2019/3/1.
- [2] Ungar, Lieberman, and Fry. “Debugging and the Experience of Immediacy”. In: *Communications of the ACM* 40.4 (1997), pp. 38–43. DOI: 10.1145/248448.248457.
- [3] Hancock. “Real-Time Programming and the Big Ideas of Computational Literacy”. PhD thesis. Massachusetts Institute of Technology, Sept. 2003.
- [4] Lieber, Brandt, and Miller. “Addressing Misconceptions About Code With Always-on Programming Visualizations”. In: *Proceedings of CHI 2014*. ACM, 2014, pp. 2481–2490. DOI: 10.1145/2556288.2557409.

- [5] Krämer, Kurz, Karrer, and Borchers. “How Live Coding Affects Developers’ Coding Behavior”. In: *Proceedings of VL/HCC 2014*. Melbourne, VIC, Australia: IEEE Computer Society, July 2014, pp. 5–8. DOI: 10.1109/VLHCC.2014.6883013.
- [6] Wilcox, Atwood, Burnett, Cadiz, and Cook. “Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems?” In: *Proceedings of CHI 1997*. Atlanta, Georgia, USA: ACM, 1997, pp. 258–265. DOI: 10.1145/258549.258721.
- [7] Hundhausen and Brown. “An Experimental Study of the Impact of Visual Semantic Feedback on Novice Programming”. In: *Journal of Visual Languages & Computing* 18.6 (2007), pp. 537–559. DOI: 10.1016/j.jvlc.2006.09.001.
- [8] Alaboudi and LaToza. “Edit - Run Behavior in Programming and Debugging”. In: *Proceedings of VL/HCC 2021*. IEEE, 2021, pp. 1–10. DOI: 10.1109/VL/HCC51201.2021.9576170.
- [9] Ingalls, Kaehler, Maloney, Wallace, and Kay. “Back to the Future: The Story of Squeak - A Usable Smalltalk Written in Itself”. In: *Proceedings of OOPSLA 1997*. ACM, 1997, pp. 318–326. DOI: 10.1145/263698.263754.
- [10] Liu and Li. “Task Complexity: A Review and Conceptualization Framework”. In: *International Journal of Industrial Ergonomics* 42.6 (2012), pp. 553–568. DOI: 10.1016/j.ergon.2012.09.001.
- [11] Fleming, Scaffidi, Piorkowski, Burnett, Bellamy, Lawrance, and Kwan. “An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks”. In: *ACM Transactions on Software Engineering and Methodology* 22.2 (Mar. 2013), 14:1–14:41. DOI: 10.1145/2430545.2430551.
- [12] Lawrance, Bogart, Burnett, Bellamy, Rector, and Fleming. “How Programmers Debug, Revisited: An Information Foraging Theory Perspective”. In: *IEEE Transactions on Software Engineering* 39.2 (Feb. 2013), pp. 197–215. DOI: 10.1109/tse.2010.111.
- [13] Rein, Ramson, Beckmann, and Hirschfeld. *Artifact for “Does Task Complexity Moderate the Benefits of Liveness? - A Controlled Experiment”*. Version 1.0. Zenodo, Sept. 2024. DOI: 10.5281/zenodo.13854016.

## 4. Published Project Results

### 4.1 Publications with Scientific Quality Assurance

- [R1] Rein, Ramson, Beckmann, and Hirschfeld. “Does Task Complexity Moderate the Benefits of Liveness? - A Controlled Experiment”. In: *The Art, Science, and Engineering of Programming* 9.1 (Oct. 2024). Open Access, pp. 1–39. DOI: 10.22152/programming-journal.org/2025/9/1.
- [R2] Rein, Beckmann, Krebs, Mattis, and Hirschfeld. “Too Simple? Notions of Task Complexity Used in Maintenance-Based Studies of Programming Tools”. In: *Proceedings of the International Conference on Program Comprehension (ICPC) 2023*. Authors’ Version PDF. IEEE, 2023, pp. 254–265. DOI: 10.1109/ICPC58990.2023.00040.
- [R3] Rein, Beckmann, Mattis, and Hirschfeld. “Toward Understanding Task Complexity in Maintenance-Based Studies of Programming Tools”. In: *Proceedings of the International Workshop on Programming Experience (PX/22) 2022*. Open Access. ACM, 2022. DOI: 10.1145/3532512.3535223.

- [R4] Rein, Beckmann, Geier, Mattis, and Hirschfeld. “Competitive Debugging: Toward Contests Promoting Debugging as a Skill”. In: *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) 2022*. Open Access. ACM, 2022, pp. 172–184. DOI: 10.1145/3563835.3567665.
- [R5] Beckmann, Rein, Ramson, Bergsiek, and Hirschfeld. “Structured Editing for All: Deriving Usable Structured Editors From Grammars”. In: *Proceedings of the Conference on Human Factors in Computing Systems (CHI) 2023*. Open Access. ACM, 2023, 595:1–595:16. DOI: 10.1145/3544548.3580785.
- [R6] Methfessel, Beckmann, Rein, Ramson, and Hirschfeld. “MμSE: Supporting Exploration of Software-Hardware Interactions Through Examples”. In: *Proceedings of the Conference on Human Factors in Computing Systems (CHI) 2024*. Open Access. ACM, 2024, 936:1–936:16. DOI: 10.1145/3613904.3642186.
- [R7] Krebs, Mattis, Rein, and Hirschfeld. “Toward Studying Example-Based Live Programming in CS/SE Education”. In: *Proceedings of the International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT) 2023*. Open Access. ACM, 2023, pp. 17–24. DOI: 10.1145/3623504.3623568.
- [R8] Stachnik, Beckmann, Rein, and Hirschfeld. “SpecTacles: Supporting Control Flow Comprehension of Software Developers in TLA+”. In: *Proceedings of VL/HCC 2024*. Authors’ Version PDF. Oct. 2024, pp. 174–186. DOI: 10.1109/VL/HCC60511.2024.00028.
- [R9] Krebs, Rein, and Hirschfeld. “Example Mining: Assisting Example Creation to Enhance Code Comprehension”. In: *Proceedings of the International Workshop on Programming Experience (PX/22) 2022*. Open Access. ACM, 2022, pp. 60–66. DOI: 10.1145/3532512.3535226.
- [R10] Rein, Flach, Ramson, Krebs, and Hirschfeld. “Broadening the View of Live Programmers: Integrating a Cross-Cutting Perspective on Run-Time Behavior Into a Live Programming Environment”. In: *The Art, Science, and Engineering of Programming* 8.3 (Feb. 2024). Open Access. DOI: 10.22152/programming-journal.org/2024/8/13.

## 4.2 Other Publications and Published Results

### Materials of Experiment 1

The artifact includes setup (environment, task materials, protocol, questionnaires), data (measurements, event sequences, questionnaire results) and analysis procedures (conversion and analysis scripts)

Rein, Ramson, Beckmann, and Hirschfeld. *Artifact for “Does Task Complexity Moderate the Benefits of Liveness? - A Controlled Experiment”*. Version 1.0. Zenodo, Sept. 2024. DOI: 10.5281/zenodo.13854016

### Materials of Task Complexity Survey

The artifact includes a list of all candidate publications considered and the intermediate data analysis tables of the publication selection and extraction of factors

Rein, Beckmann, Krebs, Mattis, and Hirschfeld. *Dataset for "Too Simple? Notions of Task Complexity used in Maintenance-based Studies of Programming Tools"*. Version 1.0. May 2023. DOI: 10.5281/zenodo.7632523

### **Library for Tracking User Interactions in Squeak/Smalltalk**

The library that was used for tracking interactions in programming tools in experiment 1 and experiment 2

Rein and Hirschfeld. *StarTrack*. Version 0.8.0. July 2023. URL: <https://github.com/hpi-swa-lab/StarTrack>

### **Materials for Conducting Debugging Contests**

The artifact includes scenarios, tasks, procedures, and event streaming server setup

Rein, Beckmann, Geier, Mattis, and Hirschfeld. *Materials for Conducting Debugging Contests*. Oct. 2022. DOI: 10.5281/zenodo.7223815

### **Materials of Experiment 2**

As the analysis and preparation of a publication for experiment 2 are ongoing, the materials for experiment 2 are not yet publicly available. We will publish the experiment materials and results, the results of the edit-run-analysis, and our generalized version of the edit-run-analysis that is applicable to a diverse range of experiments.