Implementing the Visual Debugger: Past, Present, and Future

Tim Kräuter
tkra@hvl.no
Western Norway University of Applied Sciences
Bergen, Norway

Patrick Stünkel
patrick.stuenkel@hvl.no
Western Norway University of Applied Sciences
Bergen, Norway

Adrian Rutle
aru@hvl.no
Western Norway University of Applied Sciences
Bergen, Norway

Yngve Lamo
yla@hvl.no
Western Norway University of Applied Sciences
Bergen, Norway

ABSTRACT

The visual debugger is an IntelliJ IDEA plugin that presents debug information as an object diagram for improved program understanding. Remembering our *past* plugin development, we detail the lessons learned and roadblocks we have experienced while implementing and integrating the visual debugger into IntelliJ IDEA. Furthermore, we describe recent improvements and new features added to the visual debugger, greatly improving the plugin in the *present*. Looking into the *future*, we propose solutions to overcome the roadblocks encountered during plugin development and further plans for the visual debugger.

KEYWORDS

Debugging, Visual Debugging, Visual Debugger, Visualization, IDE plugin, IDE extension, IDE Integration

ACM Reference Format:
Tim Kräuter, Patrick Stünkel, Adrian Rutle, and Yngve Lamo. 2024. Implementing the Visual Debugger: Past, Present, and Future. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

This paper details the experience of implementing, maintaining, and improving the *visual debugger* [11]. The visual debugger is available for IntelliJ IDEA and Android studio as a plugin [17], but its architecture makes it easily adaptable to other Integrated Development Environments (IDEs). The visual debugger automatically hooks into the IDE's debugging process and graphically depicts the current debug information as an *object diagram* to foster program comprehension [11].

The contributions of this paper are twofold:

(1) We describe the improvements and new features added to the visual debugger since our last publication [11], see subsection 2.3. Especially how we improved the integration of our plugin into IntelliJ IDEA.

(2) We discuss the lessons learned and roadblocks we experienced while developing the visual debugger as a plugin for IntelliJ IDEA. In addition, we propose methods to mitigate these roadblocks to achieve smoother and easier IDE integration in the future.

The remainder of this paper is structured as follows. We describe the visual debugger in (section 2), including its architecture and the improvements and new features we have implemented recently. Then, we outline the lessons learned and roadblocks we encountered during development in section 3 and how to possibly overcome them. Finally, we conclude in section 4.

2 THE VISUAL DEBUGGER

The visual debugger is an open-source IntelliJ IDEA plugin that visualizes the debugging variables as an object diagram to improve program comprehension. It is available for IntelliJ IDEA and Android Studio through the JetBrains Marketplace [10, 17] making use of the IntelliJ Platform [12]. Our plugin was integrated into IntelliJ IDEA, the Java IDE with the highest popularity, with approximately 70% market share as per the JVM Ecosystem Report 2021 [3].

Debugging information might not be present in the top-level variables and has to be obtained by digging multiple levels (following links to related objects) deep into different variables. Thus, in specific scenarios, especially when data is hierarchically structured, a graphical representation results in a faster and better understanding of the debugging information [11].

Until now, we have only received positive feedback regarding the visual debugger, which now has over 7300 downloads[1]. This marks a more than twofold increase in downloads compared to the initial release of our research paper [11] on July 21, 2022, which garnered approximately 2700 downloads.

Additional artifacts, including source code, a demonstration of the visual debugger tool, and a description of the Visual Debugging API, can be found in [8].

2.1 Description

Traditionally, debugging information is represented textually, such as in Figure 1, which shows a screenshot of the debugging variables view in IntelliJ IDEA.

[1]Last checked on the 2nd of December, 2023, see [17].

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

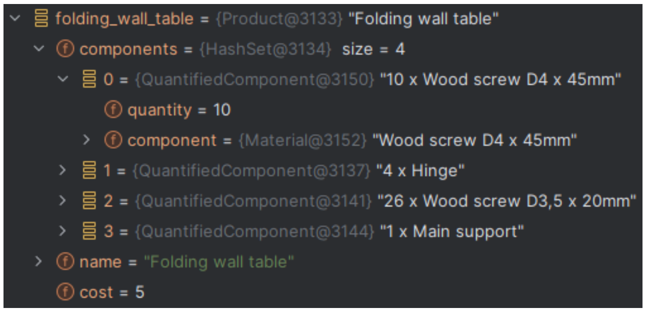

Figure 1: Variables during debugging in IntelliJ IDEA

The visual debugger will represent the same information graphically as an object diagram. Using the visual debugger is *straightforward* since it becomes available automatically during debugging in the IDE. After activating the visual debugger, it continuously visualizes the variables in the scope of the debugging session (see, for example, Figure 2). Figure 2 contains the same objects and level of detail as Figure 1. There is a little more debug information in Figure 1 due to well-written toString() methods which are used by IntelliJ IDEA. In addition, the coloring in Figure 2 highlights variable changes and additions, a new feature of the visual debugger discussed in subsection 2.3.

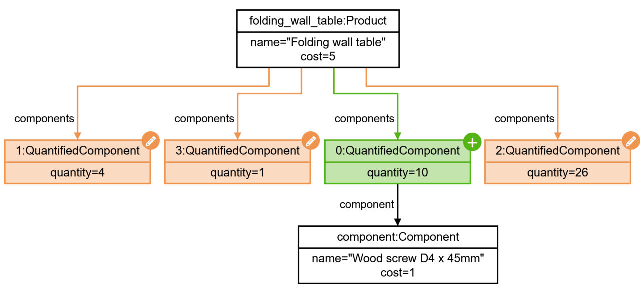

Figure 2: Object diagram of Figure 1 in the visual debugger

The graphical visualization does not replace the textual debugging view but aims to improve program comprehension during debugging[11]. Concretely, the visual debugger is *non-intrusive* since it can be used alongside the traditional textual debugging available in the IDE.

The visual debugger automatically updates the debug information just as the IntelliJ IDEA debugger whenever a user steps through the source code or a new breakpoint is reached. Moreover, children of a debugging variable can be loaded by double-clicking an object in the object diagram, similar to how it works in most textual debuggers. For example, in Figure 2, all children of the green object were loaded. The goal is to make the visual debugger *familiar* by adopting how textual debuggers work such that a transition is smooth.

More information about the visual debugger, including a typical usage scenario, can be found in [11]. In addition, a demonstration of the visual debugger is available at https://www.youtube.com/watch?v=lU_OgotweRk, and the tool can be installed using [17].

2.2 Architecture

In this section, we briefly summarize the architecture of the visual debugger. The plugin's architecture plays an important role in the improvements and new features of the visual debugger, as well as in understanding the roadblocks we discuss in section 3. The visual debugger is separated into two independent components communicating through the *Visual Debugging API* [11]. Figure 3 summarizes the architecture of the visual debugger.

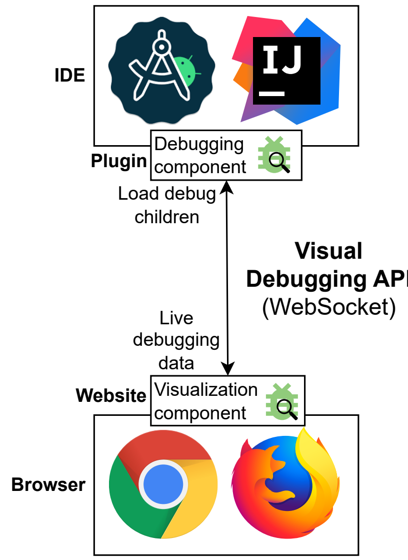

Figure 3: Visual debugger architecture

The first component, the *debugging component*, integrates with IntelliJ IDEA, and its primary function is to acquire and refresh debugging information throughout the debugging process. The debugging component makes this information available via a WebSocket server that implements our Visual Debugging API. Upon establishing a connection to the Visual Debugging API, a client receives real-time updates with the latest debugging information and can request to load children for an existing object.

The second component, the *visualization component*, portrays debugging information as an object diagram for better understanding (refer to Figure 2 for an illustration). It is implemented using web technologies and our object diagram library [9] to visualize the debug information. Additionally, it leverages the Visual Debugging API to communicate with the debugging component. Thus, it is agnostic of the IDE used for debugging and can even be reused for other programming languages than Java/Kotlin. In practice, the visualization component is hosted on a web server as part of the visual debugger plugin.

The downside to this flexibility is that the visual debugger is not entirely integrated into IntelliJ IDEA, as the visualization occurs in a browser external to the IDE. The rationale behind opting for this approach instead of native IDE integration is discussed in section 3. In addition, the next section discusses recent improvements to the visual debugger, which led to a better IDE integration despite a web-based user interface.

2.3 Improvements & New Features

We made two major improvements to the visual debugger, optimizing its integration with the IDE and expanding its applicability to more complex debugging scenarios.

Improvement (1): We integrated an embedded browser into the visual debugger panel inside IntelliJ IDEA. The embedded browser uses the Java Chromium Embedded Framework (JCEF), available by default in IntelliJ IDEA. As a result, users now have the option to use the embedded instead of an external browser. Most of the functionality of our visualization component worked out of the box with the JCEF browser. Additionally, nearly all other features, for example, exporting object diagrams to XML/SVG, were successfully implemented using JCEF APIs.

The Chromium Embedded Framework (CEF) [13] is integrated with numerous programming languages, including the Java integration JCEF. The CEF plays a crucial role in many popular applications, for example, the cross-platform Steam Client[2]. However, integrating web applications into IntelliJ IDEA is not entirely seamless, which is also highlighted by the fact that using JCEF is still an experimental feature in IntelliJ IDEA. Consequently, we give each user the choice to use the embedded or external browser. We discuss the problems we faced using JCEF and possible improvements in section 3.

Improvement (2): We enhanced the loading mechanism for debugging information inside the visual debugger. Previously, we had to pre-load and cache more debug information than was requested by the user since we could not load debug information on demand. The initial load invalidated the underlying stack frame supplied by the Java Debugging Interface (JDI). Now, we leverage the APIs provided by IntelliJ IDEA, which are a thin wrapper around the JDI. This enables us to defer loading additional debug information until explicitly required. Consequently, there is no longer a need to pre-load potentially unnecessary debug information. This optimization has improved performance, making the visual debugger applicable to more intricate debugging scenarios.

Furthermore, we enhanced the visual debugger by introducing the following two new features:

Feature (1): The visualization component of the visual debugger now *highlights changes* using colors and overlays in the object diagram. New objects and links are colored green, while changes to existing elements lead to orange coloring. Computing and highlighting the changes is enabled by default but can be switched off. Figure 2 shows the new change visualization by highlighting three changed, and one added object accordingly. A software engineer is usually most interested in the changes that occur to the objects during debugging. Our color-based visualization of changes in the object diagram makes it easier for a software engineer to see changes even when dealing with a complex debugging situation with multiple connected objects.

Changes can be calculated efficiently using unique object IDs provided during debugging. We implemented the change detection in the visualization component such that it can be reused across programming languages and IDEs [8].

Feature (2): Furthermore, the visualization component now keeps a *debug history* so that a user can see debug information from previous debugging steps. The debug history's length is configurable or can be turned off entirely. As described earlier, software engineers are most interested in how variables change during debugging. Consequently, to not only highlight differences to the previous step, we save the previous debugging information in a debug history. A software engineer can thus inspect the previously shown debug information and step as far back as he configured. The visual debugger always shows where the debug information was collected in the source code, and highlights changes compared to the previous debugging step. In addition, one can still export the debug information to an image or even edit it in our object diagram modeler [9] for documentation purposes [11].

We implemented the *debug history* in the visualization component to be independent of the used programming language and IDE. Thus, both features follow our aim of providing a reusable visualization of the debugging information set in our previous publication [11].

3 LESSONS LEARNED & ROADBLOCKS

The most important lesson we learned is that there are more features available during plugin development than what is documented. Thus, after a thorough search in the documentation, it is beneficial to ask a question in the community forum for plugin development. If one asks precise, detailed questions, one receives helpful answers quickly, making the forum an immensely valuable resource. For example, our new deferred loading of debug information discussed in subsection 2.3 was enabled due to the IntelliJ IDEA forum. The forum provides the rare opportunity to interact directly with JetBrains developers.

While developing the visual debugger plugin, we encountered two significant roadblocks. In the remaining portion of this section, we will describe these roadblocks and how to overcome them in the future.

Roadblock (1): Creating a user interface with interactive diagramming capabilities and integrating it into IntelliJ IDEA was challenging for different reasons. The IntelliJ Platform is built on Java, and plugin user interfaces utilize the Swing framework. However, we could not find a diagramming framework for Swing to implement an interactive object diagram for our visualization component. IntelliJ IDEA uses *yFiles*[3], for example, to visualize generated class diagrams from source code. For us *yFiles* was not an option because the cheapest license already carries a significant cost[4] and we do not plan to monetize our plugin.

Thus, we started with an embedded visualizer using PlantUML [1] to generate pictures of object diagrams. Finally, to allow interactions with object diagrams, we opted for the free and open-source *diagram-js* library to implement our web-based library *object-diagram-js* [9] used in the visual debugger. To summarize, the rich web ecosystem and its growing popularity among developers has led us to a web-based visualization. Integrating a web-based user interface into IntelliJ IDEA is not obvious compared to other IDEs, such as Visual Studio Code, which is built on web technologies and heavily uses web views for extensions.

[2]https://developer.valvesoftware.com/wiki/Chromium_Embedded_Framework

[3]https://www.yworks.com/products/yfiles

[4]A single license for a single developer and application currently costs 10,000 USD.

To integrate our web-based user interface into IntelliJ IDEA, we now use JCEF as described in subsection 2.3. We were unaware of this possibility for an extended period. We believe the plugin documentation regarding JCEF and web views has room for improvement to take full advantage of the rich web ecosystem. First, it should be more visible that the integration of web views is possible. Second, the integration poses challenges that can be reduced by more extensive documentation about JCEF. Currently, we are authoring a pull request to improve the plugin documentation, incorporating insights gained from our experience with the visual debugger.

Roadblock (2): Other IDEs which are less popular than IntelliJ IDEA are missing debugging-related APIs to implement our plugin. Users of the visual debugger have asked us if the plugin could be adapted to work with other IDEs and programming languages. For example, we would like to support debugging *Go* applications inside the GoLand IDE. However, compared to IntelliJ IDEA plugins, there is no API to hook into debugging processes in GoLand. Thus, it is currently not possible to adapt the visual debugger for GoLand such that it seamlessly integrates with the IDE. Our visualization component cannot operate without a debugging component that provides debugging information from GoLand. This represents a major roadblock to adopting the visual debugger for other IDEs and programming languages. One could develop the necessary APIs and debugging components for each desired IDE and programming language combination. Nevertheless, there would be significant redundancy across all these implementations.

A possible superior solution would be to utilize the *Debug Adapter Protocol* (DAP) [14]. The DAP is a sibling of the more popular *Language Server Protocol* (LSP) [15] and standardizes an abstract protocol for how a development tool communicates with concrete debuggers. The motivation is that debuggers have to be implemented only once for each language and then can be reused in different IDEs, Editors, or other tools, such as our visual debugger, see Figure 4. Since not all current debuggers will adopt this protocol, an intermediary component is envisioned to adapt an existing debugger to the DAP [14], see *debug adapters* in Figure 4.

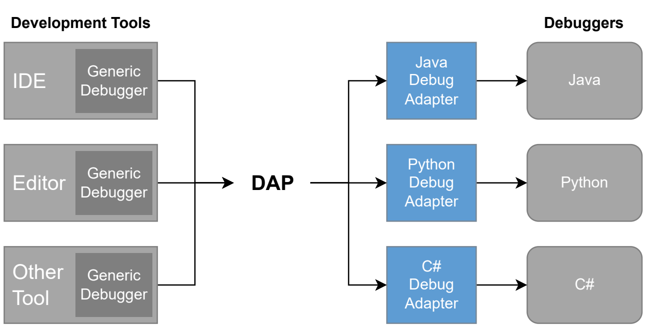

Figure 4: Debug Adapter Protocol Architecture [14]

Like our visual debugging API, the DAP is independent of development tools (as shown on the left side of Figure 4) and is also language-agnostic (as depicted on the right side of Figure 4). How JetBrains or other IDEs communicate with its integrated debuggers does not have to use the DAP. However, each IDE could provide debug adapters for the supported programming languages or integrate existing debug adapters into the IDE. In addition, using a standard protocol requires minimal documentation and leads to fewer support inquiries.

If the visual debugger also supports the DAP in the future, it can attach to the different debug adapters, which all provide the DAP, i.e., the same interface. As a result, the visual debugger becomes compatible with any combination of IDE and programming language that provides a debug adapter. For example, the visual debugger could be automatically available for more JetBrains IDEs, such as GoLand (Go), RustRover (Rust), Rider (C#), and more.

After the success of the LSP, it seems like the DAP is on track to become the next standardized development tool functionality [2, 16]. The official page of the DAP lists 11 tools supporting the DAP, 67 debug adapters, and 11 DAP SDKs as of December 2024 [14], including, for example, a Go DAP implementation maintained by Google [6]. Research is also conducted on the DAP to debug Domain-Specific Languages [5, 7].

4 CONCLUSION & FUTURE WORK

The visual debugger has increased in popularity, measured by the more than doubling of downloads of the plugin compared to our previous publication [11]. In this work, we make two contributions related to the visual debugger and the broader topic of IDE integration.

Our first contribution is represented by the improvements and new features we incorporated into the visual debugger. We have integrated our visual debugger more smoothly into the IDE by employing an embedded browser based on the Java Chromium Embedded Framework (JCEF) and improved the performance of loading debugging information. Furthermore, the visual debugger now highlights changes graphically using colors and overlays, and we implemented a debug history.

Our second contribution is detailing the experience gained by implementing the visual debugger. We described two major roadblocks hampering tighter IDE integration of our plugin. First, integrating web-based user interfaces that use the extensive and popular web ecosystem is not trivial. Using the JCEF makes this integration possible, but its visibility and documentation should be improved. Second, not all IDEs offer debugging-related APIs such that we could integrate our plugin. To make debugging functionality uniformly accessible for plugin development, we propose to utilize the standardized Debug Adapter Protocol (DAP).

In future work, we aim to adapt the visual debugger for other IDEs and code editors such as GoLang, Eclipse [4], and Visual Studio Code since this has been requested by our users. We aim to implement the DAP to minimize the integration effort and development cost for the different IDEs.

Furthermore, it would be interesting to study the usability of the visual debugger and its impact on program comprehension. It seems interesting to find out in which scenarios visual debugging is more effective than textual debugging and vice versa. Since the visual debugger is *not* meant to replace the textual debugger, one should also include a combination of both textual and visual debugging in an empirical study.

REFERENCES

[1] Arnaud Roques. 2023. PlantUML. https://plantuml.com/.

[2] Dominik Bork and Philip Langer. 2023. Language Server Protocol: An Introduction to the Protocol, Its Use, and Adoption for Web Modeling Tools. *Enterprise Modelling and Information Systems Architectures (EMISAJ)* 18, 9 (Sept. 2023), 1–16. https://doi.org/10.18417/EMISA.18.9

[3] Brian Vermeer. 2021. JVM Ecosystem Report 2021 | Snyk. https://snyk.io/jvm-ecosystem-report-2021/.

[4] J. desRivieres and J. Wiegand. 2004. Eclipse: A Platform for Integrating Development Tools. *IBM Systems Journal* 43, 2 (2004), 371–383. https://doi.org/10.1147/sj.432.0371

[5] Josselin Enet, Erwan Bousse, Massimo Tisi, and Gerson Sunyé. 2023. Protocol-Based Interactive Debugging for Domain-Specific Languages. *The Journal of Object Technology* 22, 2 (2023), 2:1. https://doi.org/10.5381/jot.2023.22.2.a6

[6] Google. 2023. Go Implementation of the Debug Adapter Protocol. https://github.com/google/go-dap.

[7] Pierre Jeanjean, Benoit Combemale, and Olivier Barais. 2021. IDE as Code: Reifying Language Protocols as First-Class Citizens. In *14th Innovations in Software Engineering Conference*. ACM, Bhubaneswar, Odisha India, 1–5. https://doi.org/10.1145/3452383.3452406

[8] Tim Kräuter. 2023. ICSE-2024: Artifacts. Zenodo. https://doi.org/10.5281/ZENODO.10210019

[9] Tim Kräuter. 2023. Object-Diagram-Js. Zenodo. https://doi.org/10.5281/ZENODO.10018182

[10] Tim Kräuter. 2023. The Visual Debugger Tool. Zenodo. https://doi.org/10.5281/ZENODO.10018177

[11] Tim Kräuter, Harald König, Adrian Rutle, and Yngve Lamo. 2022. The Visual Debugger Tool. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Limassol, Cyprus, 494–498. https://doi.org/10.1109/ICSME55016.2022.00066

[12] Zarina Kurbatova, Yaroslav Golubev, Vladimir Kovalenko, and Timofey Bryksin. 2021. The IntelliJ Platform: A Framework for Building Plugins and Mining Software Data. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, Melbourne, Australia, 14–17. https://doi.org/10.1109/ASEW52652.2021.00016

[13] Marshall Greenblatt. 2023. Chromium Embedded Framework (CEF). https://github.com/chromiumembedded/cef.

[14] Microsoft. 2023. Debug Adapter Protocol. https://microsoft.github.io/debug-adapter-protocol/.

[15] Microsoft. 2023. Language Server Protocol. https://microsoft.github.io/language-server-protocol/.

[16] Jonas Rask, Frederik Madsen, Nick Battle, Hugo Macedo, and Peter Larsen. 2020. Visual Studio Code VDM Support. In *Proceedings of the 18th International Overture Workshop*. arXiv, Online, 35–49. https://doi.org/10.48550/arXiv.2101.07261

[17] Tim Kräuter. 2023. Visual Debugger - IntelliJ IDEs Plugin | Marketplace. https://plugins.jetbrains.com/plugin/16851-visual-debugger.

Received 7 December 2023