The Visual Debugger Tool

Tim Kräuter
Høgskulen på Vestlandet
Bergen, Norway
tkra@hvl.no

***Abstract*—Debugging is an essential part of software maintenance and evolution since it allows a software developer to analyze a program step by step. Understanding a program is required to fix potential flaws, alleviate bottlenecks, and implement new desired features. Thus, software developers spend a large percentage of their time validating and debugging software, resulting in high software maintenance and evolution cost. We aim to reduce this cost by providing a novel visual debugging tool to software developers to better support them during debugging. Our debugging tool visualizes program information graphically as an object diagram and is fully integrated into the popular Java development environment IntelliJ IDEA.**

***Index Terms*—Debugging, Visual Debugging, Visual Debugger, IntelliJ IDEA Plugin, Software Maintenance/Visualization**

I. Introduction

Debugging is an essential part of software maintenance and evolution since it allows a software developer to analyze a program step by step during execution. Nowadays, debugging tools are integrated with every modern Integrated Development Environment (IDE) and are generally seen as indispensable. Debugging is used to understand program control- and data flow such that a software developer can locate and fix reported bugs or extend the program to implement new desired features. Thus, debugging is crucial for software maintenance and evolution, and software developers spend between 35 and 50 percent of their time validating and debugging software [1]. Consequently, 50-75 percent of the total budget of software development projects is used for debugging, testing, and verification [1]. We aim to reduce this cost by providing a novel debugging tool to software developers to better support them during debugging. Reduced time spent on debugging can be used to implement new features, i.e., creating value for customers.

Traditionally program information is represented in a textual manner during debugging (see Figure 2 in section II). Debugging tools integrated with IDEs, such as IntelliJ IDEA and Eclipse, show a set of top-level variables directly contained in the current program scope. However, the desired program information is often not present in the top-level variables but spread out on lower levels of potentially different variables. Thus, in specific scenarios, a graphical representation results in a faster and better understanding of the shown program information. We have developed a tool that visualizes the current program information graphically as an object diagram for better program comprehension. Our open-source tool called the *visual debugger* is integrated with IntelliJ IDEA[1], which is the most popular Java IDE according to the JVM Ecosystem Report 2021 [3]. Compared to other tools, our visual debugger is optimized for industrial use since it is straightforward, lightweight, and can be used alongside the traditional textual debugger. In addition, the tool's architecture enables the reuse of the visualization component in other debugging tools. A demonstration of our tool is available online [4][2].

The remainder of this paper is structured as follows. We describe the visual debugger tool in detail (section II) before explaining the tool architecture (section III). Finally, we discuss related work in section IV and conclude in section V.

II. Tool description

We will describe our tool using the parts list model shown in Figure 1. A parts list describes the decomposition of Products into sub-products and basic Materials. Given a parts list, one can calculate the monetary cost and materials needed to construct one or more pieces of the described product[3].

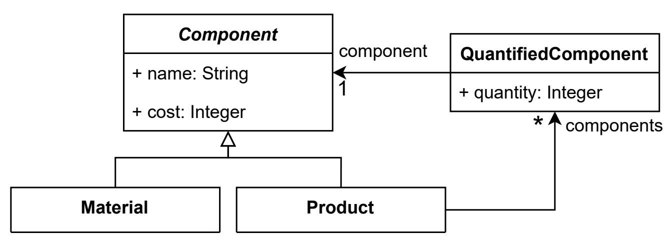

Fig. 1. Parts list class diagram

Figure 2 shows objects during debugging in IntelliJ IDEA conforming to the parts list model. The default debugger uses a textual representation for program information.

We have unfolded the substructure of the folding wall table object to see its components and first component in detail. If one is not only interested in one attribute inside one object, but rather the whole object world using the textual representation is cumbersome.

Consequently, research on visual debugging began with the goal of fostering program comprehension. Our tool is one of many visual debugging tools, but we aimed for excellent usability by seamlessly integrating our tool in the debugging

[1]The tool is available through the JetBrains Marketplace [2].
[2]Tool demonstration: https://www.youtube.com/watch?v=lU_OgotweRk
[3]This example is inspired by the course on information systems taught by Michael Löwe at the University of Applied Sciences FHDW, Hanover.

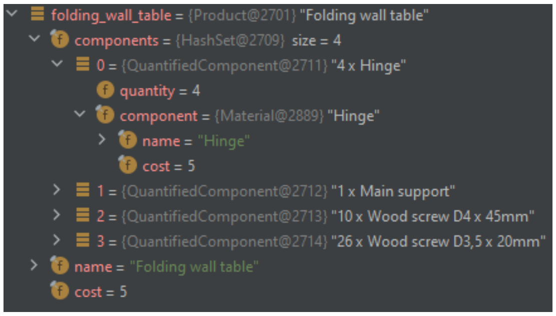

Fig. 2. Variables during debugging in IntelliJ IDEA

process of the IntelliJ IDEA. In addition, our tool is straightforward and non-intrusive, i.e., it complements textual debugging. The goal of our tool is to make debugging during software development as efficient as possible to increase software developer productivity.

Using our visual debugger tool, we obtain the object diagram[4] shown in Figure 3, which contains the same objects and level of detail as Figure 2.

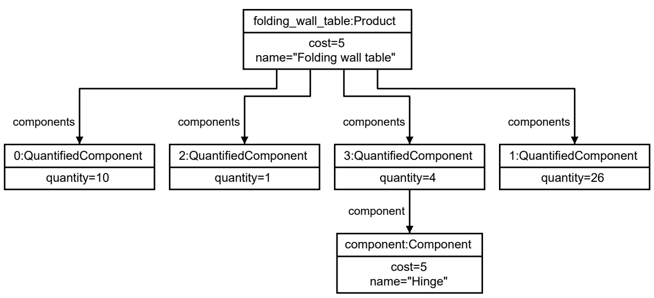

Fig. 3. Visual Debugger visualization comparable to Figure 2

The visual debugger tool continuously visualizes the variables in the scope of the debugging session as an *object diagram*. The visualization starts automatically when the first breakpoint is reached during debugging if the visual debugger tool is activated. Thus, if desired, a software developer can use the visual debugger alongside the textual debugging view. The visualization is always up to date since we listen to the events generated by a debugging session in IntelliJ IDEA. Consequently, we update the visualization whenever a new breakpoint is reached, or a user steps through the program code.

Textual debugging views only show the root objects (directly referenced in the debugging session) without attributes when debugging is started. Similarly, we do not visualize all objects linked to the root objects, but we allow the user to configure a *visualization depth*. The visualization depth describes how many links starting from the root objects should be followed to find objects for the initial visualization.

[4] Additional artifacts including source code, a demonstration of the visual debugger tool and a description of the Visual Debugging API can be found at [4].

Afterward, one can explore objects further by double-clicking them in the visualization just as in the textual debugger. For example, in Figure 3, one quantified component was explored further. The visualization is browser-based and implemented in a standalone *visualization component*, which automatically layouts the object diagram using the Eclipse Layout Kernel. In addition, we provide a visualization based on PlantUML embedded in IntelliJ IDEA. However, it is not possible to explore objects inside the embedded visualization since PlantUML provides static Unified Modeling Language (UML) diagrams.

The visual debugger tool currently has 1804 unique downloads[5] and positive reviews. It consists of the debugging and the visualization component, which we will describe in more detail in the tool architecture section. Both components are open-source[4] and, when combined, result in the visual debugger tool.

III. Tool Architecture

First, the *debugging component* integrates with IntelliJ IDEA by automatically hooking into all started debugging processes of the IDE. The goal of the debugging component is to obtain the current debugging information from IntelliJ IDEA and pass it on to the visualization component. In addition, the debugging component offers a method to load detailed information for individual objects in the current debugging scope, as described earlier. The debugging component is written in Java, and its code quality and security are continuously checked using static code analysis based on SonarCloud and unit tests.

Second, the *visualization component* represents the debugging information as an object diagram to ease program understanding. Moreover, it allows interaction to load additional debugging information for the currently shown objects. The visualization component is *browser-based* (JavaScript) and relies on a fixed *Visual Debugging Application Programming Interface (API)*. Consequently, we could implement a debugging component for a different IDE, such as Eclipse, and reuse the visualization component. Furthermore, the visualization component is independent of the programming language which is debugged, and can potentially be reused to debug different object-oriented programming language.

The *Visual Debugging API*[4] is based on *WebSocket* to allow live updates about changes in the debugging information, see Figure 4.

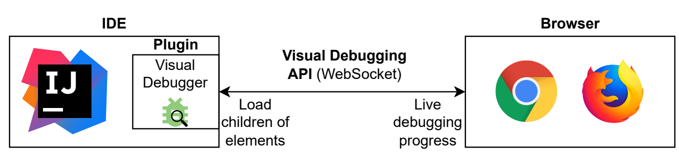

Fig. 4. Communication through the Debugger API

Initially, a browser connects to the WebSocket server hosting the Debugger API, for example, the server included in our

[5] Last checked on March 8, 2022, see [2].

Visual Debugger plugin. Afterward, the browser is updated in real-time about new debugging information due to debugging actions in the IDE, such as hitting a breakpoint or jumping to the next line in the source code. In addition, the visualization component allows a user to interact with the visualization to load all direct children of shown objects.

Debugging information, i.e., object diagram exchange, is standardized by an XSD schema [4]. Figure 5 depicts the metamodel for object diagrams realized by the schema.

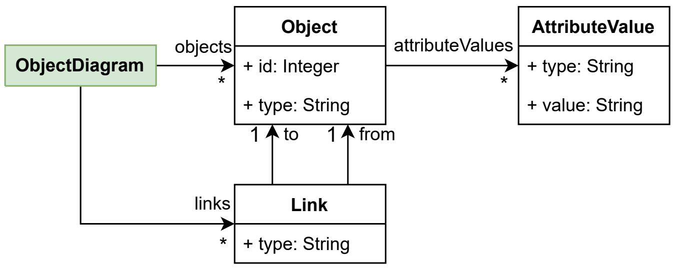

Fig. 5. Object diagram metamodel

The ObjectDiagram is the root element in the schema (highlighted in green) and contains a set of Objects and Links. Objects and Links have a type, i.e., name of a class or association. In addition, each Object has a unique id provided by the debugger and a set of attributeValues, which have a primitive type and value modeled as strings.

Besides debugging, the visualization component provides two export features. First, one can export object diagrams during debugging as an SVG file. This can be useful if an undesired program state has been reached and should be documented in a bug tracking system. Second, diagrams can be exported as an XML file that can be used to load and edit them in the object diagram modeler, for example, to show the actually desired program state. The object diagram modeler is an open-source tool to create object diagrams in the browser [5].

IV. Related work

Visual debugging has been researched since the 90s [6]–[9], but most of the resulting tools are outdated. We will now describe recent visual debugging tools and compare them to our tool.

Java Interactive Visualization Environment (JIVE) is a plugin for the Eclipse IDE [10]–[12]. It provides interactive Java program execution visualization at different levels of granularity. The program state is visualized as a UML object diagram, while the call stack is represented as a UML sequence diagram. JIVE is tightly coupled to the Eclipse IDE and does not integrate with Eclipses debugger but rather is a debugging environment on its own. This approach is a significant difference to our tool, which integrates with the debugging tool of the IDE. It makes JIVE powerful but complex since it is hard to understand what is happening in the multiple views provided by JIVE. Compared to JIVE the visual debugger tool focuses only on object diagram visualization of the current program state, which makes it lightweight and straightforward to use. In addition, our tool decouples debugging and visualization, such that it can be adopted to different IDEs even based on other object-oriented programming languages than Java.

A plugin called *Java Visualizer* has been developed for the IntelliJ IDEA [13]. It visualizes the call stack and objects contained in the Java heap as a box-and-pointer diagram during a debugging session. However, even in simple scenarios, the visualized call stacks are long since all objects from the Java heap are visualized and not only the variables in the debugging scope. This leads to much noise in the visualization, especially if one is only interested in the current objects, i.e., the current system state. In contrast, our tool only shows relevant information and allows users to load more information if needed.

In [14], the authors describe a tool to debug distributed applications. It can connect to multiple Java virtual machines and show the retrieved objects separately in an object diagram or combine the same objects from different JVMs using object identifiers or other properties. The tool is also tightly integrated with the Eclipse IDE and tackles the problem of debugging distributed applications, which we do not address. However, we could not find and test the tool by ourselves. In the future, we could incorporate these ideas by allowing multiple debugging components (one for each application) to connect to one visualization component. The visualization component can then show the different debugging views separately or combined as described in [14].

JAVAVIS is a standalone tool to help students understand program execution in Java [15]. It makes use of object- and sequence diagrams to represent program behavior. However, it is not integrated with modern IDEs such as Eclipse or IntelliJ IDEA. Our tool can help students learn Java or object-oriented program execution in general, but we currently do not provide a sequence diagram visualization.

V. Conclusion & future work

The main contribution of this paper is the new open-source visual debugging tool, which differs from previously created tools regarding the following three aspects. First, it is integrated with IntelliJ IDEA, a modern and popular IDE for Java software development. According to JVM Ecosystem Report 2021, over 70% of JVM developers use IntelliJ IDEA [3]. In addition, the plugin received good feedback and was downloaded nearly 2000 times already[5]. Second, the visualization part of the tool is independent, such that it can be reused in other visual debugging tools. For example, one could develop a plugin for Eclipse IDE or Visual Studio Code in the future. Third, we aimed for excellent usability of our tool alongside present debugging tools. Thus, it automatically starts when debugging in IntelliJ IDEA and can be used straight away without any configuration. Moreover, we only show a subset of the debugging information in the debugger right away and allow the user to reload more relevant information similarly to the widely adopted textual debuggers.

We plan to improve and extend the tool in multiple ways in the future. First, we want to do more field testing using our tool to gather feedback on the usability and features of the plugin. This should lead to continuous improvement of the tool and greater tool use, which leads to more feedback from practitioners. Especially, the scalability of the tool when debugging large software systems must be investigated. The scalability of our visual debugger should be similar to the scalability of present textual debuggers, such that our tool can be adopted in the industry. Second, we plan to implement visual debuggers for other IDEs and object-oriented programming languages by reusing our visualization component. Third, we plan to reuse significant parts of our tool to debug executions of behavioral models. Not only source code can be executed and debugged. Behavioral models, for example, UML state-machines, Petri-Nets, or Business Process Modeling Notation (BPMN) processes come with clearly defined execution semantics [16]. When these models are used in Model-driven engineering (MDE), it can be beneficial to execute and debug them directly.

REFERENCES

[1] D. H. O'Dell, "The Debugging Mindset: Understanding the Psychology of Learning Strategies Leads to Effective Problem-Solving Skills." *Queue*, vol. 15, no. 1, pp. 71–90, Feb. 2017.

[2] "Visual Debugger - IntelliJ IDEs Plugin — Marketplace," https://plugins.jetbrains.com/plugin/16851-visual-debugger.

[3] "JVM Ecosystem Report 2021 — Snyk," https://snyk.io/jvm-ecosystem-report-2021/, Jun. 2021.

[15] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic program visualization with object and sequence diagrams using the java debug interface (JDI)," in *Software Visualization*, S. Diehl, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 176–190.

[4] "Artifacts - ICSME-2022," https://github.com/timKraeuter/ICSME-2022.

[5] "Object diagram modeler," https://github.com/timKraeuter/object-diagram-modeler, Mar. 2022.

[6] R. A. Baeza-Yates, G. Quezada, and G. Valmadre, *Visual Debugging and Automatic Animation of C Programs*. WORLD SCIENTIFIC, Nov. 1996, vol. 7, pp. 46–58.

[7] D. F. Jerding and J. T. Stasko, "Using visualization to foster object-oriented program understanding," Georgia Institute of Technology, Tech. Rep., 1994.

[8] S. Mukherjea and J. T. Stasko, "Toward visual debugging: Integrating algorithm animation capabilities within a source-level debugger," *ACM Transactions on Computer-Human Interaction*, vol. 1, no. 3, pp. 215–244, Sep. 1994.

[9] D. R. Hanson and J. L. Korn, "A simple and extensible graphical debugger," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '97. USA: USENIX Association, 1997, p. 13.

[10] J. K. Czyz and B. Jayaraman, "Declarative and visual debugging in Eclipse," in *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange - Eclipse '07*. Montreal, Quebec, Canada: ACM Press, 2007, pp. 31–35.

[11] J. K. P., S. Jayaraman, B. Jayaraman, and S. M, "Finite-state model extraction and visualization from Java program execution," *Software: Practice and Experience*, vol. 51, no. 2, pp. 409–437, Feb. 2021.

[12] "JIVE: Java Interactive Visualization Environment," https://cse.buffalo.edu/jive/.

[13] "Java Visualizer - IntelliJ IDEs Plugin — Marketplace," https://plugins.jetbrains.com/plugin/11512-java-visualizer.

[14] A. Koch and A. Zündorf, "Graphical debugging of distributed applications - using UML object diagrams to visualize the state of distributed applications at runtime," in *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, ser. MODELSWARD 2015. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, 2015, pp. 223–230.

[16] T. Kräuter, "Towards behavioral consistency in heterogeneous modeling scenarios," in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2021, pp. 666–671.