JOHANNES KEPLER
UNIVERSITY LINZ

Author
Katrin Anna Kern, BSc

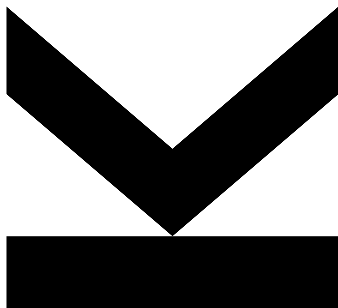
Submission
Institute for System Software

Thesis Supervisor
a.Univ.-Prof. Dipl.-Ing. Dr. Herbert Prähofer

Co-Supervisor
Dr. Markus Weninger

August 2023

JavaWiz – A Visualization Tool for Software Development Education



Master's Thesis

to confer the academic degree of

Diplom-Ingenieurin

in the Master's Program

Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Abstract

Novice programmers often have difficulty understanding the execution of their programs. When are which variables created and changed? How are they passed to functions? How are objects stored and referenced? Without detailed knowledge of how the execution environment works, many concepts remain too abstract. As a result, questions remain unanswered and understanding is poor, which can lead to poorer learning outcomes and frustration.

Visualizing the execution in a way that is appropriate for beginners can support the learning process. Visualizations should show the states during execution and thus provide an understanding of the effects of instructions. In this thesis, the tool JavaWiz (short for "Java Wizard") is presented for this purpose. JavaWiz visualizes the execution of a Java program in a form suitable for programming beginners. It thus provides several views of the program's behavior, including a tabular representation of the executed statements together with the current variable values, and a representation of the Java Virtual Machine stack and heap.

Kurzfassung

Programmieranfänger haben oft Schwierigkeiten, die Ausführung ihrer Programme zu verstehen. Wann werden welche Variablen erstellt und geändert? Wie werden sie an Funktionen übergeben? Wie werden Objekte gespeichert und referenziert? Ohne detailliertes Wissen darüber, wie die Ausführungsumgebung funktioniert, bleiben viele Konzepte zu abstrakt. Infolgedessen bleiben Fragen unbeantwortet und das Verständnis ist schlecht, was zu schlechteren Lernergebnissen und Frustration führen kann.

Die Visualisierung der Ausführung auf eine für Anfänger geeignete Weise kann den Lernprozess unterstützen. Visualisierungen sollen die Zustände während der Ausführung zeigen und so ein Verständnis für die Auswirkungen von Anweisungen vermitteln. In dieser Arbeit wird zu diesem Zweck das Tool JavaWiz (kurz für "Java Wizard") vorgestellt. JavaWiz visualisiert die Ausführung eines Java-Programms in einer für Programmieranfänger geeigneten Form. So bietet es

mehrere Ansichten des Programmverhaltens, darunter eine tabellarische Darstellung der ausgeführten Anweisungen zusammen mit den aktuellen Variablenwerten sowie eine Darstellung des Stacks und Heaps der Java Virtual Machine.

Table of Content

Contents

Abstract **i**

Kurzfassung **i**

1 Introduction **1**

2 Background **5**

2.1 eInformatics@Austria 5

2.2 MOOC Programming with Java 5

2.3 Software Development 1 at JKU 6

2.4 Program Visualizations 6

3 Visual Teaching Aids **10**

3.1 BlueJ 10

3.2 Greenfoot 11

3.3 JAVAVIS 11

3.4 Online Java Tutor 13

3.5 Jeliot 14

3.6 Jsvee 15

4 JavaWiz **18**

4.1 Users 18

4.2 How JavaWiz Works 18

4.3 Download and Installation 19

4.3.1 Visual Studio Code Extension 19

4.3.2 Web Version 20

4.4 Feature Overview 21

4.5 Visualization Components 24

4.5.1 Desk Test 24

4.5.2 Stack and Heap . . . 26
4.5.3 Console . . . 28

5 Implementation **29**

5.1 Frameworks and Libraries . . . 29
5.1.1 Vue.js . . . 29
5.1.2 Vuex . . . 33
5.1.3 D3.js . . . 35
5.1.4 d3-graphviz and DOT . . . 37
5.1.5 JDI . . . 38
5.1.6 Java Parser . . . 39
5.1.7 Monaco Editor . . . 39
5.2 Architecture . . . 39
5.3 Communication . . . 41
5.4 Processing the Trace . . . 44
5.5 Editor . . . 47
5.6 Desk Test . . . 49
5.7 Stack and Heap . . . 55
5.8 Console . . . 59
5.9 Extension . . . 61

6 Conclusion and Future Work **63**

Literature **66**

1 Introduction

A core problem in learning programming is a lack of understanding how the computer actually works, i.e., it is a "black box" [32]. If this black box, or rather the *notional machine* it contains, is not explained thoroughly, students will make assumptions about it that might not always be correct. This can lead to confusion and a poor understanding of what is actually happening, as the executed code often results in a program state that is different from the students' expectations. Du Boulay described understanding this mapping between code and execution results as "[...] just as hard as trying to understand how a car engine works from a diagram in a text book." [29].

Thus, it is of the utmost importance to provide students with explanations and practical examples of the correct behaviour of a program. This process can be supported in mainly three ways: metaphors, a simpler machine and visualizations [32].

Metaphors Metaphors in the context of computer science education are a way of making abstract concepts more tangible by explaining them through concepts students already know. This is also the crux - there is usually no 1:1 mapping between the abstract concept and the known one, so it still leaves room for misconceptions. Kohn and Komm [32] explained this problem with the common "variables = box" metaphor: some students might assume that a box can contain multiple values at the same time.

Simpler machines Instead of using a computer as we know it, the machine can be simplified in a way that students can fully grasp its capabilities. A form of this is making the machine act as a "computing agent". One of the most famous examples of such an agent-based system is Scratch [21]. Users can add multiple agents to a canvas and attach instructions to each, which are then visibly executed. While systems like that are certainly useful for a playful introduction and gaining basic understanding, they are limited to the provided building blocks which usually do not encompass a full programming language. Introductory programming classes

at university level usually use a specific programming language from the beginning to teach general concepts.

Visualizations Visualizations in various forms can be used to support computer science education. However, in a traditional classroom setting, they mostly consist of drawings on a whiteboard or animated slides. Both are tedious and time-consuming to make and inflexible. Small modifications, for example, to answer a follow-up question of students, can be hard to visualize as both drawings and slides might become hard to read. To mitigate those issues, visualization tools can be used. Some tools have their own learning curve though, taking away valuable time from the actual subject. They also have to fit the courses' programming language and style of teaching. For example, there are not a lot of Java tools available and none of them fit the needs of how programming is currently taught at JKU (see Section 2).

Let us assume we moved successfully past the initial core concepts - our students know that a "box" (variable) only contains a single value and they understand how they can use loops and conditional expressions to move a visual computing agent. Our black box still contains further *threshold concepts*, a term coined by Meyer and Land [34] to describe potentially blocking concepts in education. These concepts are usually difficult to understand, they integrate concepts or transform the perspective of a student [34]. Two examples in computer science are object-oriented programming and references, which were identified as threshold concepts in an empirical study by Boustedt et. al [27]. To support students in moving past these concepts, it makes sense to use the above-mentioned methods. For example, classes can be explained with the "blueprint" metaphor - a class is the "blueprint" of a house, an object is then the actually built house, building it is called instantiating and so on. A pointer could be described as an entry in the content table of a book, pointing to a specific page in that book. In an agent-based system, the agents are essentially objects with their own variables and methods.

However, these two methods have their limits. How would a static variable fit into the blueprint metaphor, for example? How would one describe the concept of

pointers in Scratch? This is where program visualizations come into play and can provide the missing pieces.

In this thesis, therefore, a visualization tool will be presented. The tool is named JavaWiz ("Java Wizard") and should aid both teachers and students in programming education. Its application area ranges from the very first code lines to object-oriented programming, i.e., it can support roughly the content covered in a beginner's course at university level. Multi-threading, stream operations or being able to handle larger projects are out of scope.

From the discussion in this chapter, we can already define a few general requirements that we focus on:

- Ready-to-use. The tool should be simple and fast to use – for teachers to reduce preparation time, for students to avoid an additional learning curve. Ideally, it is self-explanatory.
- Complete. The tool should cover what beginners usually find difficult, so the points where other methods are lacking can be covered and there is no need to switch mid-course to a different, advanced tool or resort to drawing by hand.
- Real. The tool should not hide or oversimplify the features of the programming language to avoid misconceptions that might lead to confusion down the road.
- Fits the teaching strategy. The tool should fit the teaching style of the course. At JKU, they start with imperative programming using Java (see Section 2 for more details).

Structure The structure of this thesis is as follows:

- Section 2 explains the context of the project.
- Section 3 discusses existing visualization tools and their advantages and disadvantages.

- Section 4 contains a high-level description of JavaWiz features.
- In Section 5, the actual implementation is described in detail.
- Section 6 concludes the thesis with describing the current usage scenarios of the tool and introducing further ideas.

2 Background

This section outlines the context in which the tool was created, which is important for understanding why certain features were implemented. It also includes a brief introduction into the teaching methods used in programming classes at JKU and the online course in which the tool will be used.

2.1 eInformatics@Austria

The work has been conducted in the context of the project eInformatics@Austria [6]. eInformatics@Austria is an Austrian project led by the Technical University of Vienna and with participation of several other Austrian universities. The goal is to provide university-level Massive Open Online Courses (MOOCs) for computer science education. Altogether, there will be seven MOOCs with topics like computational thinking, computer architecture, algorithms and data structures or software development. The MOOCs will be provided on the platform iMooX [9].

The eInformatics team at the Institute for System Software (SSW) at JKU is responsible for creating the course "Programmieren mit Java" (Programming with Java). The course teaches imperative programming in Java, i.e., elementary control structures, simple data types, arrays, string handling to finally classes, objects and methods, but without inheritance.

2.2 MOOC Programming with Java

The MOOC, available on iMooX [9] starting from October 2023 (only in German), is mostly following the structure and content of the JKU course "Software Development 1". It is self-paced and split up into two parts (Programming with Java 1 and 2). There are 11 chapters in total, with each chapter containing multiple videos, exercises and quizzes. In contrast to the traditional class at JKU, the lectures are split up into small parts (max. 15 minutes) and are followed with videos of the exercises and their solutions. At the end of each chapter, a quiz is used to check the progress. In general, the course uses "Mastery Learning" [28], which means that students are required to go through the material sequentially; only moving

on when they have successfully “mastered” a part. The self-paced manner of the course supports this, as there are no externally imposed deadlines and the course stays online (with the exceptions of updates). The exercises increase in difficulty within a chapter and require knowledge from previous chapters. Some exercise themes will recur throughout a topic or chapter, providing a familiar environment to apply newly acquired skills.

In contrast to a traditional course, there are no lecturers actively teaching and discussing with the students. To enable interactive engagement with the material, JavaWiz is provided to students as a Visual Studio Code [22] plugin. Usually, programming courses have a rather steep learning curve and high drop-out rates. Thus, with the MOOC, we aim to provide an environment where students can comfortably learn programming at their own pace and university students have additional practice material to solidify their knowledge.

2.3 Software Development 1 at JKU

The university course at JKU is a mandatory course in the curricula of several Bachelor’s degrees (Computer Science, Mechatronics, Electronics and Information Technology). The course is split up into a 90 minutes lecture and an 90 minutes exercise class per week over 12 weeks (3 + 3 ECTS workload). The lecture provides the theoretical foundation and the exercise is used to repeat important concepts, show live examples and discuss the weekly assignments.

The course is held in an imperative-first style, starting with variables and data types before moving on to control structures and finally to object-oriented programming.

2.4 Program Visualizations

In the lecture as well as the exercise classes of Software Development 1, some visual teaching aids are already used. This includes drawings on the blackboard, animated slides and static graphics or diagrams in the slides. While these visualizations surely help with understanding, they are time-consuming in class and

during preparation and are always limited to specific examples. Small, live modifications, e.g., for explaining details or in response to questions, are tedious to make. Hence, one goal of JavaWiz is to support teaching by providing the same visualizations lecturers currently use in their classes. The most commonly used visualizations are:

- The “desk test” (Figure 1), also known as desk check, which is an informal way of checking the correctness of smaller parts of a program. Usually, desk tests are done with pen and paper – hence the name – by drawing a table with the variables as columns and the lines of code (or their numbers) as rows. After “executing” every line mentally, the current values of the variables are written down. This helps with understanding the execution sequence, for example during learning different loop types, and with finding errors.
- Simple diagrams in slides or on the blackboard are used for introducing object-oriented concepts. Usually, objects are depicted as boxes and references as arrows (Figure 2). To save space and time, details like data types or strings actually pointing to a character/byte array are often omitted.
- For programming with arrays, they are usually depicted as boxes with separate slots for their values, often combined with reference arrows (Figure 3) and indices written below or above the slots.

```
    ...
5   int[] arr = {1, 4, 3};
6   int sum = 0;
7
8   for(int i = 0; i < arr.length; i++) {
9       sum += arr[i];
10  }
    ...
```

line	sum	i	arr[i]
5	-	-	-
6	0	-	-
8	0	0	1
9	1	0	1
8	1	1	4
9	5	1	4
8	5	2	3
...			

Figure 1: Example of a desk test for a simple loop.

```
Person p1 = new Person("John Doe", 35);
Person p2 = p1;
```

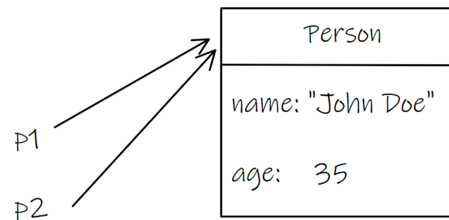


Figure 2: Example of a typical simple object visualization.

```
int[] arr = {1, 4, 3};
```

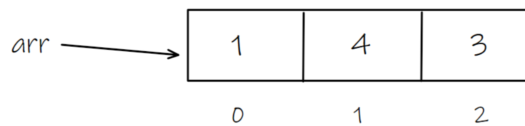


Figure 3: Example of a typical simple array visualization.

3 Visual Teaching Aids

In this section, several existing tools and approaches for program visualizations targeted at beginners are introduced.

3.1 BlueJ

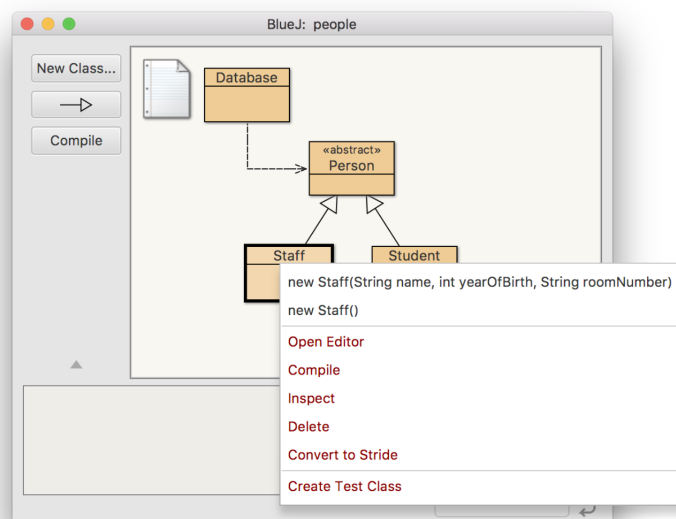


Figure 4: BlueJ window with class operations. Source: [3]

BlueJ [2] is an educational development environment using an object-first approach for learning Java. It comes in the form of a desktop tool and is available for most Java platforms. The central part of the tool is a canvas, on which classes can be created. From the classes, objects can be created by interacting with the context menu (see Figure 4) and dialogue windows, adding them visually to the "object bank" (the lower part of the window). With the same interaction style, methods can be called on objects. It is also possible to edit the source code directly. The tool also features a "code pad", which allows the user to execute small snippets in a REPL (read-execute-print-loop). Moreover, it supports threads.

While BlueJ is an excellent and widely-used tool, it does not meet our requirements. First and foremost, it is not imperative-first. It could be used later in the course, when object-oriented programming is introduced, but that would require switching and therefore fails the “Completeness” requirement. The 38-page tutorial also suggests that the tool might not be the fastest and simplest to master. However, it does not hide any relevant language features and can even be used for more advanced topics like multi-threading.

Another thing to note is that the tool is more an alternative to a classic IDE (which beginners often have trouble with) and less a tool to explore how memory works, for example.

3.2 Greenfoot

Greenfoot [8] is another educational desktop tool for learning Java and uses – like BlueJ – an object-first approach. It could be described as a mix between BlueJ and Scratch - the interface and interaction styles (canvas with visual object representation, context menus, dialogue windows, full code editor) are similar to BlueJ, while its focus is on creating graphic-heavy “scenarios” with a “world” and “actors”, similar to Scratch. The user defines the logic of the actors and the world, while the tool takes care of the actual drawing [33]. This enables the users to tackle visually appealing and otherwise complex programs early on, for example games (Figure 5) and simulations.

However, due to its object-first nature and graphics focus, it is - like BlueJ - not suitable for our needs.

3.3 JAVAVIS

JAVAVIS [13] is another desktop tool for learning Java, featuring object diagrams and sequence diagrams (Figure 6). The tool was published in 2002 - to our knowledge, it has not been further developed and cannot be downloaded. The implementation is based on the Java Debug Interface (JDI) [18].

It is closer to what we have imagined for this project: first of all, it is imperative-

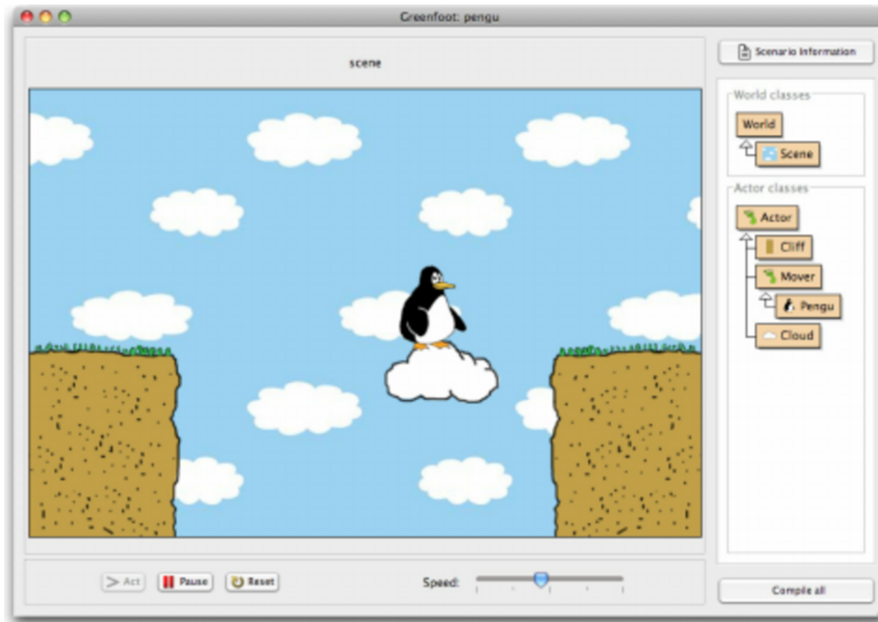


Figure 5: Greenfoot Programming Environment. Source: [33]

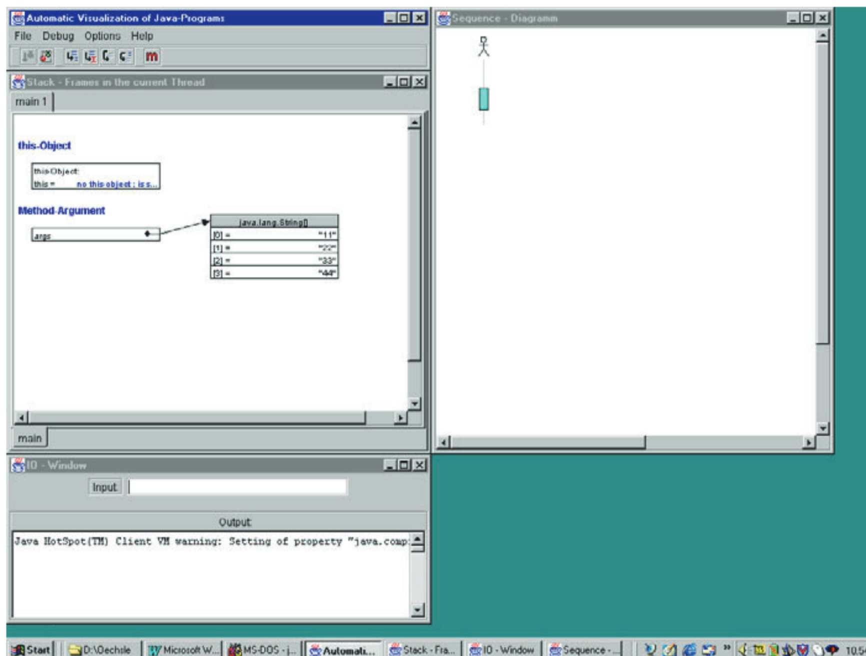


Figure 6: JAVAVIS main window. From: [13]

first. It provides an object diagram window showing current local variables and objects on the heap, including type and visibility information. In addition, it shows a sequence diagram of the currently running program, with support for threads. User input can be provided via a text field.

Again, we see some limitations: the stack frames are visualized in separate tabs - even by only looking at the screenshots in the paper, this seems to make it hard to have a good overview of the program. The source code and currently executed line is not visible alongside the diagrams, which is sub-optimal for learning how to map code to resulting program states.

3.4 Online Java Tutor

Online Java Tutor [10] is a web-based tool for running and inspecting Java programs (Figure 7). The website also provides the same tool for Python (the original tool, see [30]), C, C++ and JavaScript. The web-based approach has some advantages compared to the desktop tools we have seen so far. It requires no installation, which lowers the entry barrier and enables usage on computers with user restrictions, such as library or school computers. The state of the program is encoded in the URL and can thus be shared with teachers or other students. It is also embeddable, which makes it possible to include the tool on websites such as online textbooks.

Other noteworthy features are:

- The objects can be moved across the canvas by drag-and-drop, which can be useful for programs with many objects.
- Individual objects and object fields can be hidden by entering their name into a text box and pressing a button to update the visualization.
- The visualization can be customized by changing, for example, the length of arrows.
- The tool also highlights the next line to be executed and the line that was just executed and allows jumping to particular points in the program.

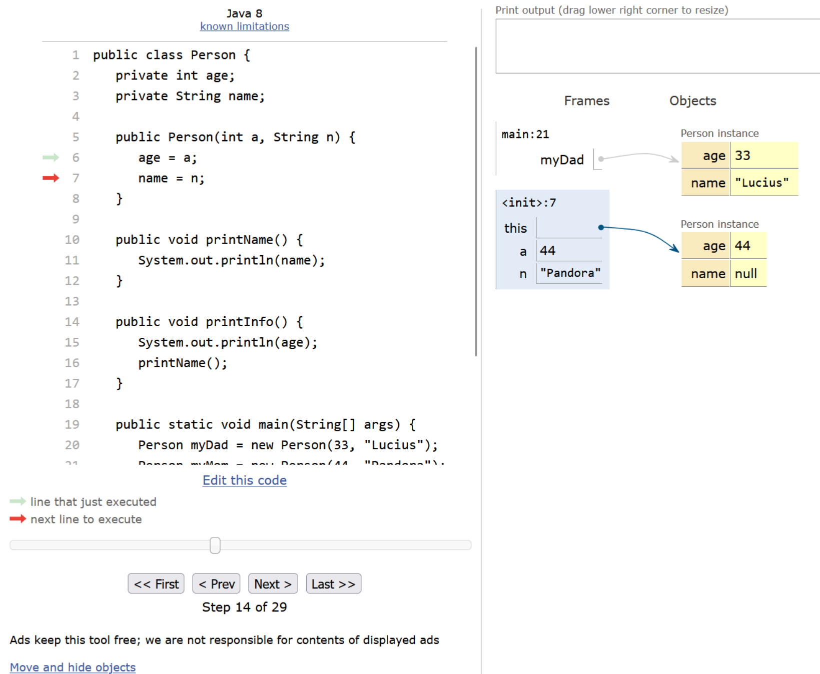


Figure 7: Java Tutor. Screenshot taken from [10].

However, there are a few limitations to the Java tool [19]. It does not support user input - in our MOOC and the course at JKU, user input via the standard input is used early on. It also does not show the declared type of objects, only the type at runtime, which is an important part of understanding object-oriented programming. The creators also note that of all the tools on their website, it is the slowest and most likely to crash. And while it is possible to hide objects, the process seems rather cumbersome, opposed to a way of collapsing and expanding objects by clicking.

3.5 Jeliot

Jeliot [26] was one of the first tools for animating programs. In terms of user interface design, it resembles an improved version of JAVAVIS. There is a source code panel, highlighting the next line that will be executed, and a panel with four different areas (Figure 8). These are: method area, expression evaluation area,

constant area and an instance and array area. In the method area, each stack frame will have its own block. From the block, local variables are pointing to elements in the instance/array area. Variables are visualized as little boxes with another box inside for the value of the variable. The evaluation area will show how expressions are evaluated, for example, if a loop has the upper bound `n - 1`, with n being 5, the area will show the calculation `5 - 1 = 4`. The users can navigate through the program by clicking a "step" button, or start an animation by pressing "play". Input can be provided through a dialogue box, output will be displayed in a text box beneath the main panel. In a separate tab, the call tree is shown.

There are a few things that prevent us from using Jeliot. First of all, the website states that the newest version supports concepts from Java 5 and that the newest update was from 2007 [15]. In the course at JKU, many newer language features are used. It also has the side effect that the user interfaces' look-and-feel is quite outdated. Other minor issues are that newer stack frames are blocking the view of older ones and that there is no way of hiding or moving elements.

3.6 Jsvee

Jsvee [37] is a library supporting the creation of program animations. While not being a ready-made tool for students, it is still worth to mention. For example, it makes it possible to create program animations at an expression evaluation level (Figure 9). The educational debuggers or IDEs we have seen so far always execute line by line, however, with Jsvee, the execution of a line can be dissected into fetching values and evaluating expressions. This level of detail is currently not the focus for our tool, but a potential feature to keep in mind for future versions.

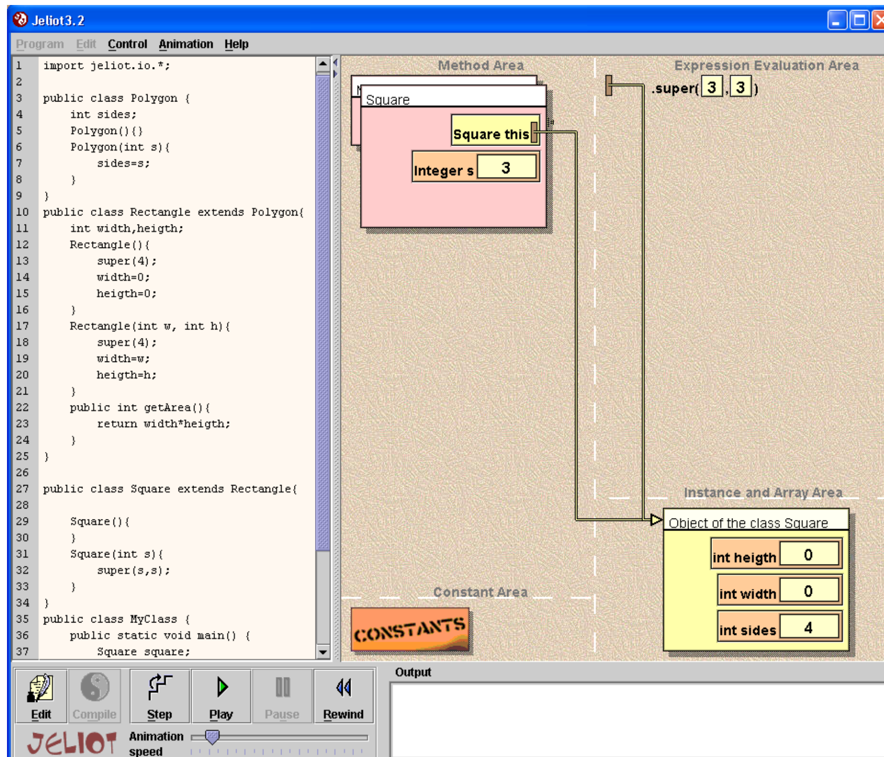


Figure 8: Jeliot. Taken from [15].

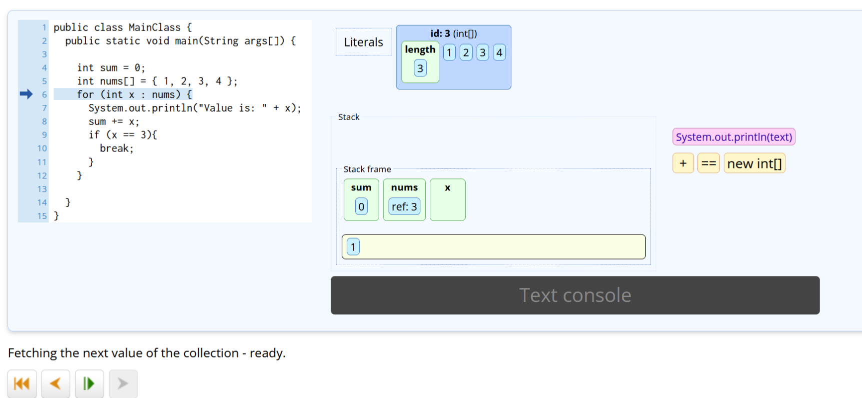


Figure 9: Jsvee. Screenshot taken from [11].

4 JavaWiz

We have now seen a plethora of different visualization tools. With the requirements from the introduction chapter in mind (ready-to-use, complete, real and fits the teaching strategy), we can look at the user groups JavaWiz targets and its concrete features.

4.1 Users

From the students' perspective, JavaWiz needs to cater both to university/school students and self-learning students. While the first group would be able to obtain help from peers and lecturers, the second group is on its own. Both types of students might use JavaWiz for solving exercises given by courses they attend or for repeating examples from the material. They might also use it for exploring concepts on their own. The focus of both functionality and user interface considerations is therefore: as simple and self-explanatory as possible.

For lecturers and teachers, the simplicity is less important, but learning to make full use of the tool should be quick nevertheless. Another thing to consider is in-classroom use, as the settings can vary greatly. The tool should work on unusual screen sizes/resolutions (smartboards or video projectors, for example) and zoom levels just as well. For in-classroom use, the startup and compile time should be kept to a minimum too.

4.2 How JavaWiz Works

The structure and data flow of the tool is described in more detail in Section 5.2. For now it is enough to grasp the general idea shown in Figure 10. The tool consists of a backend that takes care of the compilation of user code and starts a debugger. The user can then step through the program while the backend records the execution results line by line. These trace states are sent to the frontend, where they are processed and forwarded to the individual visualization components.

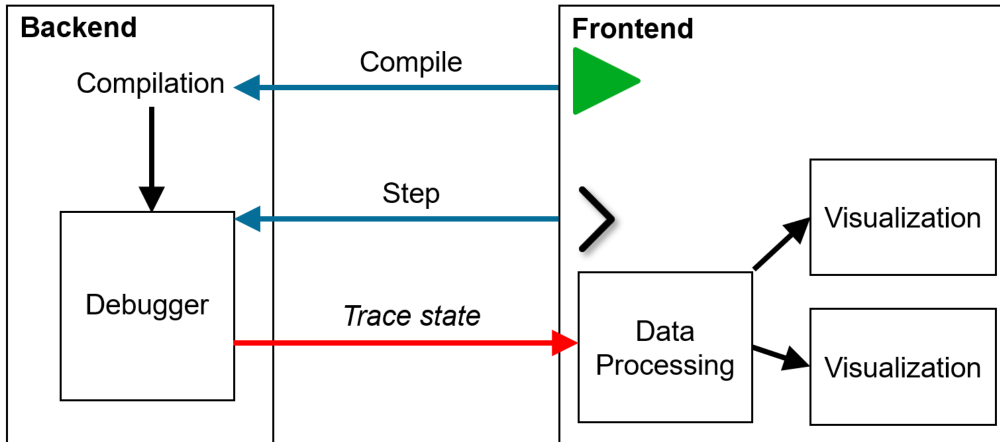


Figure 10: JavaWiz overview.

4.3 Download and Installation

4.3.1 Visual Studio Code Extension

The VS Code Extension version of the tool can be downloaded from the VS Code Marketplace (see Figure 11) [23]. To run the extension, at least JDK 17 is required. Other dependencies (listed on the Marketplace site under "Dependencies") will be installed automatically with the extension. After installation, there should be the link "Run in JavaWiz" over the main method in a Java file. A click on this link

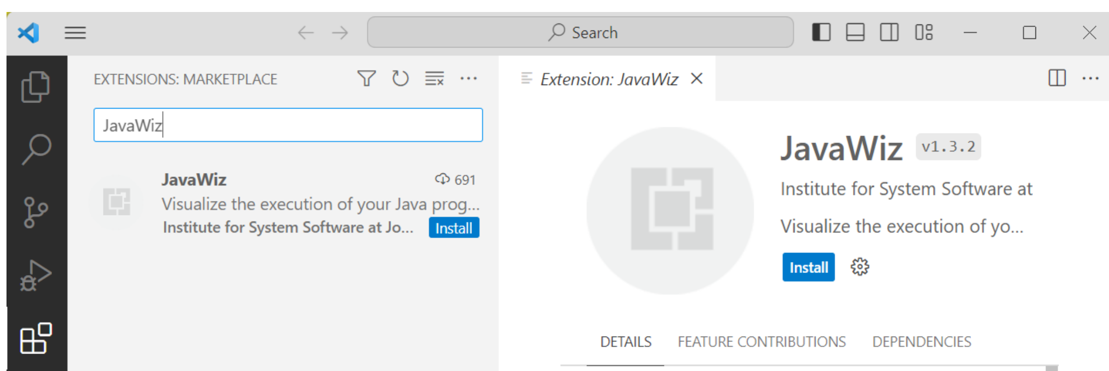


Figure 11: JavaWiz in the Visual Studio Code Marketplace.

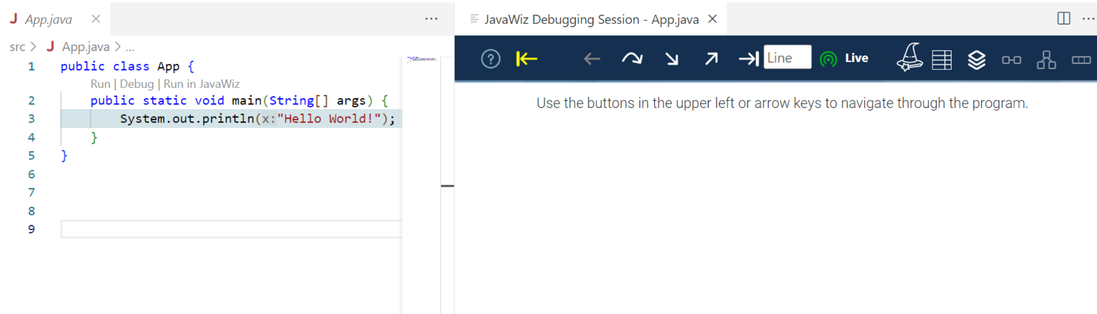


Figure 12: JavaWiz VS Code extension after starting.

will start the tool in a web view panel next to the source code and trigger the compilation of the code (see Figure 12).

4.3.2 Web Version

Besides the VS Code extension, there is a web version of JavaWiz. It runs in a browser, but needs the backend to be started manually first. The browser tool is the version used in the presentations in this thesis (more on the difference between the versions in Section 5.9). It is available through the SSW Git repository [14]. The project contains Gradle build and run scripts for both frontend and backend (see `README.md` in project root). IntelliJ users can alternatively use the included run configurations. As soon as the backend is up and running, the frontend can be opened and should connect to the backend on its own. If it was opened before the backend was running, there will be a "Connect" button on the left upper corner of the tool. The compilation can then be started by clicking on the green start button (see "Toolbar" in the following section).

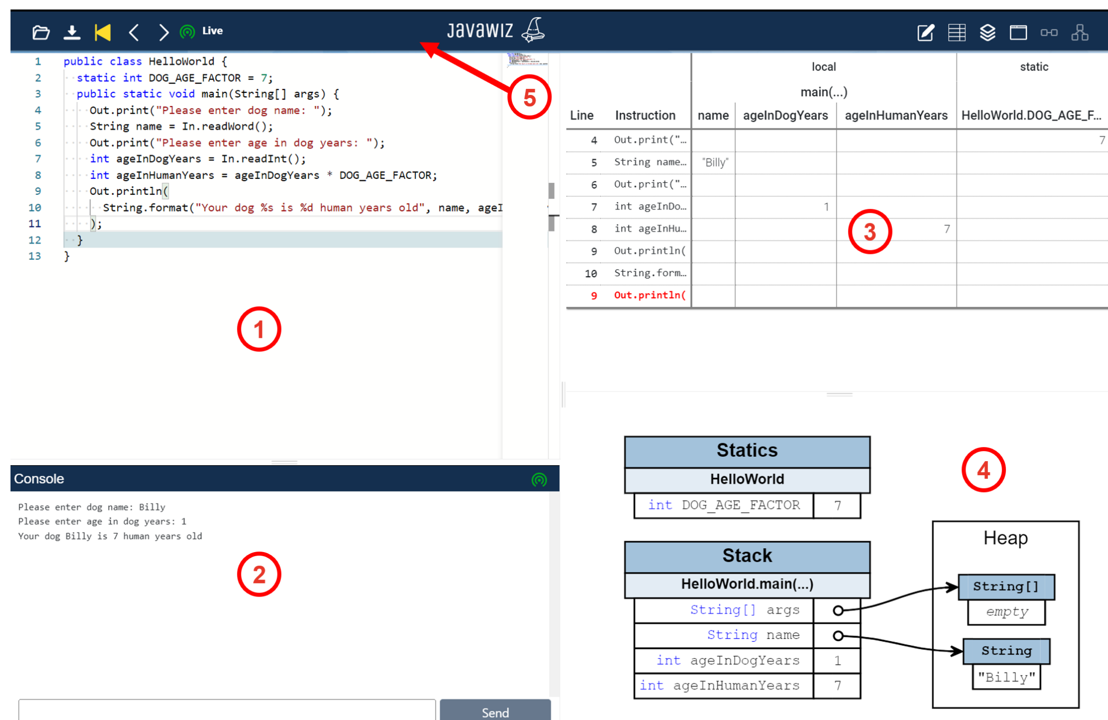


Figure 13: Overview of the JavaWiz components.

4.4 Feature Overview

Before diving into the implementation details, a high-level overview of the JavaWiz features is given (Figure 13).

① **Code editor**: The editor allows the user to write their code. While the program is executing, two highlights are provided: one for highlighting the next line to be executed, and one synced with the currently hovered line in the desk test – the same line is highlighted in the editor to help the user with orientation in the code.

② **Console**: The console is responsible for showing output and allowing to input data. Output from print statements and runtime exceptions will be shown here. Compile errors are shown in the notifications instead (see paragraph "Notifications").

(3) **Desk test**: The desk test is the digital version of the method introduced in Section 2.4. It shows the executed statements together with the variable values in a table. The idea here is to only fill in a cell if the variable value has changed at this particular execution point. It is supposed to help learners understand when and where variable values are actually updated. In JavaWiz, we extended the basic version by providing additional headers, i.e., the variables are grouped by methods and their location. Additionally, the evaluated value of conditional expressions is tracked and shown in the table, which is especially useful for teaching and learning control structures.

(4) **Stack and heap visualization**: This visualization provides a view of the current stack and heap state in a graph-like format. Object nodes include the data type, name and value of variables and objects. Pointers are depicted as arrows. Because even smaller programs can cause a pretty big heap, objects can be hidden and large arrays and objects can be shortened. Changes are highlighted in red.

(5) **Toolbar**: The toolbar provides user controls (start, reset, step back and step forward), visual controls (hide individual components), an indicator of the current program state ("live" and "replay") and controls for saving/opening code examples. Hovering any of the symbols will reveal a tool tip with the description and a keyboard shortcut.

▶ The start button takes the current code editor content and tells the backend to compile and run it.

⏮ The reset button will appear if the compilation and start of the debugger was successful and can be used to quickly recompile the same program, i.e., start from the beginning.

› The "next" button – like the "back" button – will appear after successful compilation and allow the user to step forward through the code. It acts as a "step into" for all user-defined code.

The "back" button allows the user to step back through previous trace states. When stepping forward again, the user will not produce new trace states until they reach a line that has not yet been executed. Stepping back also triggers a change in the "live"-indicator in the toolbar.

The "live"-indicator will show the current program state. The green symbol means that the next line to be executed is a new line, the yellow symbol means that previous states are being "replayed".

The icons on the right side of the toolbar can be used for hiding individual components. This button shows/hides the desk test component.

This button shows/hides the stack and heap component.

This button shows/hides the editor component.

This button shows/hides the console component.

The "open" button can open `.java` files and `.json` files in the format specified in `src\assets\configSchema.json`. The "save" button will save the code together with the information which components are hidden in the defined JSON schema (for preparing, for example, code examples where only the desk test and the editor should be shown to the user initially).

Notifications: To keep the user informed about the program state, notification overlays are shown for successful or failed compilation (Figure 14), connecting to the debugger or failed connection, and if the program potentially expects input (for example, when the user tries stepping even though the program has halted).

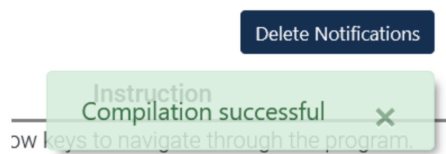


Figure 14: Notification when compilation was successful.

4.5 Visualization Components

4.5.1 Desk Test

As mentioned in the overview, the desk test is supposed to help in the very beginning of learning programming. It is useful for showing the result of statements and how control structures work.

Figure 16 shows the state of the desk test after partially executing the program in Figure 15. The desk test shows all already executed lines. In this example, the line `System.out.println(msg)` is the next line to be executed (highlighted in blue in the editor) and thus not yet shown in the desk test. Hovering a line in the desk test will highlight the same line in orange in the editor and vice versa. The just executed line is highlighted in the table with red color and bold font. If the table grows too big for its container, scrollbars will appear both horizontally and vertically. The first two columns are fixed, scrolling horizontally is therefore possible without losing track of the corresponding line number and content. In a table with a vertical scrollbar, stepping will cause the container to always scroll down to the newest line. To avoid visual clutter and reduce the need to scroll, data types and method parameters are omitted in the headings.

The table has three areas: locals, statics and conditions. Statics that are not changed anymore, like in Figure 16, will have a column, but their value will not be visible. This is because the desk test starts with the first line in the main method, i.e., where statics have already been initialized. They can be inspected in the stack and heap visualization instead. Condition columns have the expression as their header. While variable cells only contain something after their value has changed, conditions are shown after each evaluation.

Arrays are displayed as a little box with the datatype in it. This box can be hovered and will show the indices and the content of the array. Strings are shown as a simple string, not as char or byte arrays, and will be cut off after a certain length.

Objects are not a target use case of the desk test and therefore have no special representation. The table cell will just show the heap reference number.

```
1   public class App {
2     final static int A_NUMBER = 4;
      Run | Debug | Run in JavaWiz
3     public static void main(String[] args) {
4       int[] numbers = new int[] {1, A_NUMBER};
5
6       int sum = 0;
7       int i = 0;
8       while (i < numbers.length) {
9         sum += numbers[i];
10        i++;
11      }
12
13      String msg = "The sum is: " + sum;
14      System.out.println(msg);
15    }
16  }
17
18
19
```

Figure 15: Desk test code example.

		local				static	Conditions
		main					
Line	**Instruction**	**numbers**	**sum**	**i**	**msg**	**App.A_NUMBER**	**i < numbers.length**
4	int[] numbers = n…	int[]					
6	int sum = 0;		0				
7	int i = 0;			0			
8	while (i < number…						true
9	sum += numbers[i];		1				
10	i++;			1			
8	while (i < number…						true
9	sum += numbers[i];		5				
10	i++;			2			
8	while (i < number…						false
13	String msg = "The…				"The sum is: 5"		

Figure 16: Desk test execution result.

4.5.2 Stack and Heap

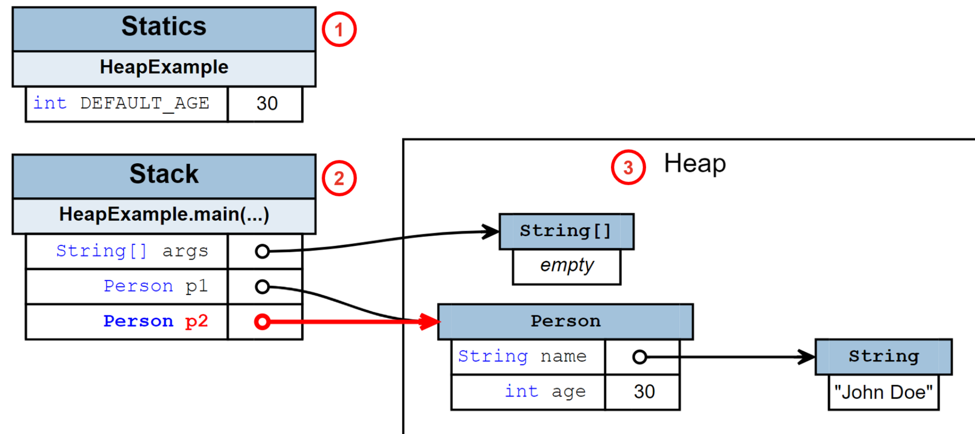


Figure 17: Example of the stack and heap visualization.

The stack and heap visualization should support students in understanding object-oriented programming. There are three parts in this visualization component (see Figure 17):

1. Static variables
2. Stack
3. Heap

The statics section has a header for each class with its static variables underneath. In the stack, these headers consist of the class name and method name. The left part of the variable tables shows the data type and variable name. The right part either shows the primitive value or a reference arrow pointing to the corresponding object on the heap. The heap is displayed as a big rectangle. Its objects have a header with the data type in it. The rest of the table depends on the object type:

- Strings are again simplified and shown as text, not as char or byte arrays (see "John Doe" in Figure 17).

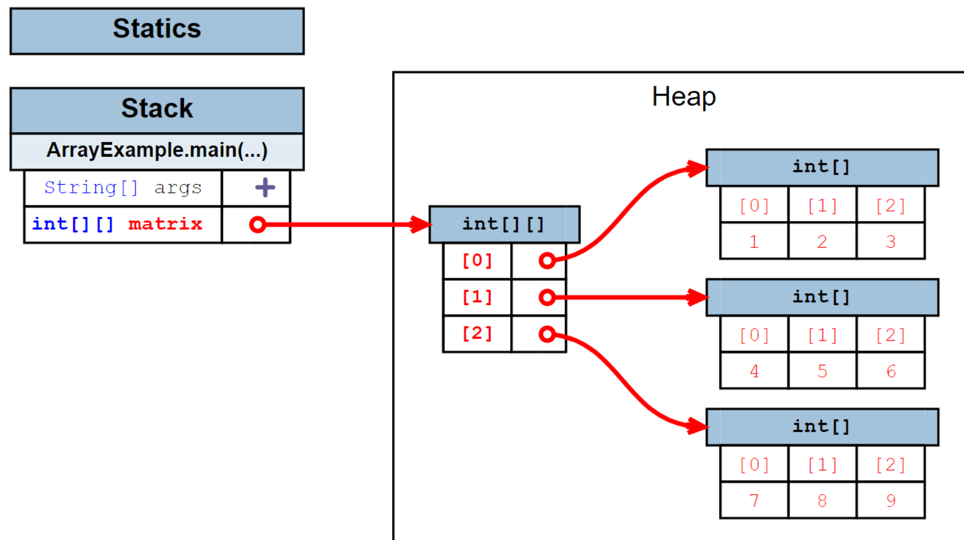


Figure 18: Two-dimensional array in the heap visualization.

- Arrays generally have the indices displayed in brackets. Primitive arrays grow horizontally, reference arrays vertically (see Figure 18. This makes multi-dimensional arrays much more readable.
- Objects and their fields are displayed in the same way as the variables on the stack, with data type and name on the left and the value (or reference arrow) on the right (see `Person` in Figure 17).

The start of a reference arrow, displayed as a circle, is clickable to hide objects the user does not want to see. After clicking, the circle will transform into a plus (see `args` in Figure 18). If there are multiple arrows pointing to an object, it will only disappear if all arrows pointing to it have been hidden. Large arrays and objects appear collapsed from the start and can be expanded if necessary. Stepping and therefore rerendering the visualization will not expand the hidden parts again.

If anything changes in the program, the variable, reference arrows and changed values will be highlighted in red and bold font. In Figure 18, for example, the user just stepped over the line where `matrix` was initialized.

```
1  public class HelloWorld {
2    public static void main ( String [] args ) {
3      Out.println("Hello World!");
4      int x = In.readInt();
5      int y = x/0;
6    }
7  }
```

Console

```
Hello World!
5
java.lang.ArithmeticException: / by zero
        at HelloWorld.main(HelloWorld.java:5)
```

Send

Figure 19: Console example.

4.5.3 Console

The console shows the output from print statements and runtime exceptions. It consists of a box where the output is shown, a text input field and a button (see Figure 19). The button will only be activated if the program expects input in the "live" mode (more details in Section 5). In "replay" mode, input will not be repeated, i.e., the input provided in the first run will be shown.

5 Implementation

In this chapter, the implementation of JavaWiz is described. First, the used technologies are introduced. Then, the architecture and the data flow are explained in detail. The rest of the chapter describes the individual components.

5.1 Frameworks and Libraries

5.1.1 Vue.js

Vue.js [24] is a component-based JavaScript framework. Similar to Angular [1] and React [20], it can be used to create dynamic Single-Page-Applications (SPA). In JavaWiz, Vue is used for splitting the application into components and managing the data flow between the components.

Vue is JavaScript-native, but can be programmed with TypeScript. We decided to switch to TypeScript later in the project as the usage of typed interfaces reduced bugs tremendously.

The components can be created with two different APIs: the Options API, which leans more towards a class-based style and the Composition API, which is more flexible. In this project, we used the Options API.

The structure, behaviour and graphics of a Vue component are usually defined in a single `*.vue` file. The HTML is defined within the `<template>` tags. Similar to other web frameworks, the template can be enriched with *text interpolation (1)*, *directives (2)* and *bindings (3)*. The features are shown in the example in Listing 1 and Listing 2. The result can be found in Figure 20.

Text interpolation is a way of injecting dynamic content into the template. Directives also help with that. For example, a `v-if` renders an element based on the result of its expression. A `v-for` is useful for rendering iterable structures. Bindings can help with dynamically changing CSS classes or managing the state of input fields.

Styles are defined with CSS within the `<style>` tags and can be scoped if necessary. The behaviour is defined within the `<script>` tags. This part contains the component definition.

The most important parts of this definition are:

- `props`: Props are attributes that can be passed from the parent component to the child component. If the parent property changes, the changed value will be passed down to the child, but the child is not supposed to mutate it (unidirectional data flow). In Listing 2, the `ChildComponent` is passed an array from its parent `App`.

- `data()`: A function returning the initial state of the component. The properties can be accessed with `this.propertyName`. In Listing 2, `newItem` is initialized with an empty string.

- `computed`: This is the area where *computed properties* can be defined. Computed properties are essentially getter functions that keep track of their dependencies. In the example in Listing 2, `nrOfItemsText` will be updated if the list length changes.

- `watch`: In this section, *watchers* are defined. Watchers listen to a property and invoke a callback function when it changes. In our example, `item added` will be logged to the console when the computed property `nrOfItemsText` changes.

- `mounted()`: `mounted` is one of Vue's lifecycle hooks and the only one needed in this project. It is called after the initial rendering has finished and DOM nodes were created, i.e., when the component was "mounted". This allows developers to execute additional code for the component initialization without running into problems with, for example, undefined DOM nodes. In this example, it is used to register an event handler on the button, allowing the "enter" key to trigger the button and to empty the input field when it was pressed.

- `methods`: In the last section, user-defined methods can be added. Here, it is used to add a new item to the list. The method is bound to the button with `v-on:click`.

```
<template>
  <h3 class="heading">{{ header }}</h3> <!-- (1) Text
   interpolation -->
  <ChildComponent :list="list" />
</template>

<script>
import { defineComponent } from 'vue';
import ChildComponent from './ChildComponent.vue';

export default defineComponent({
  name: 'App',
  components: {
    ChildComponent
  },
  data () {
    return {
      header: "Vue example",
      list: ["nr 1", "nr 2"]
    }
  },
})
</script>

<style>
    .heading {
        color: blue;
    }
</style>
```

Listing 1: App.vue

```
1  <template>
2    <div v-if="list.length > 0">{{ nrOfItemsText }}</div>
3    <ul>
4      <li v-for="item in list">{{ item }}</li> <!-- (2) Directives
       -->
5    </ul>
6    <input id="input-field" v-model="newItem" /> <!-- (3) Bindings
       -->
7    <button id="input-button" v-on:click="() => addItem()">Add Item
       </button>
8  </template>
9
10 <script>
11 import { defineComponent } from 'vue';
12
13 export default defineComponent({
14   name: 'ChildComponent',
15   props: {
16     list: {
17         type: Array
18     }
19   },
20   data () {
21     return {
22       newItem: ""
23     }
24   },
25   computed: {
26     nrOfItemsText: function () {
27         return "List with " + this.list.length + " items"
28     }
29   },
30   watch: {
31     nrOfItemsText: function () {
32         console.log("item added")
33     }
34   },
35   mounted () {
```

```
36      document.getElementById("input-field").addEventListener("keyup
      ", (event) => {
37          event.preventDefault()
38          if (event.key === "Enter") {
39              document.getElementById("input-button").click()
40              this.newItem = ""
41          }
42      })
43    },
44    methods: {
45      addItem: function () {
46        this.list.push(this.newItem)
47      }
48    }
49  })
50  </script>
```

Listing 2: ChildComponent.vue

Vue example

List with 2 items

- nr 1
- nr 2

nr 3 | Add Item

Figure 20: Rendering result of Listing 1 and Listing 2

5.1.2 Vuex

Vuex [25] is a state management library for Vue.js. JavaWiz has multiple components with partially shared state. To make the code more maintainable, a Vuex store was created. Essentially, shared states are pulled out of the components into

a global store. This could look like Listing 3. Here, we save `sharedValue` as a global state. Under `mutations`, a function for assigning a new value to it is defined.

```
import { createStore } from 'vuex'

export default createStore({
  state: {
    sharedValue: 0,
  },
  mutations: {
    changeSharedValue (state, newValue) {
      state.sharedValue = newValue
    }
  }
})
```

Listing 3: Vuex store example.

Listing 4 shows how the store would be used in a Vue component. The value of a state can be retrieved with `this.$store.stateName`. Changing the value is done by calling `this.$store.commit(...)`, passing a string with the name of the mutation and the value.

```
[...]
computed: {
  sharedValue: function () {
    return this.$store.state.sharedValue
  }
},
methods: {
  someMethod: function () {
    [...]
  this.$store.commit('changeSharedValue', value)
  }
}
[...]
```

Listing 4: Vuex store usage.

5.1.3 D3.js

d3.js (Data-Driven Documents) [4] is a JavaScript-based framework for creating dynamic and interactive visualizations. d3.js adopts a declarative programming style, where data is bound to DOM elements. In JavaWiz, the first version of the desk test was created with d3.js.

d3.js is best explained by an example. In Listing 6, we have a data array with some strings we want to display in a list. Listing 5 shows the HTML template for the unordered list which is filled by the `rerender()` function.

The `rerender()` function is shown in Listing 6. It works as follows:

- The `select` function selects the DOM element with the id `#shopping-list`. This is the element it will manipulate.
- Then, the `selectAll` function selects all `li` children. Initially, the size of this selection is zero.
- With the `data()` call, we tell d3 to bind the passed data to the selection.
- The `enter()` call and the subsequent `append()` tell d3 to append a new `li` element.
- The `style()` function sets the color of new `li` elements.
- The `text()` function uses the data to define the content of the `li` element, in this case they are identical.
- The `transition()` together with the `duration()` and the `style` function define an animation where the item appears within 500ms.

Similar to the `enter()` call, changes to existing elements can be handled with `update()`. `exit()` makes it possible to define behaviour of elements that we remove from the dataset, for example fade-out animations.

With these basic building blocks and a few other d3.js functions, an HTML table is constructed from the trace state to create the desk test.

```html
[...]
  <body>
    <ul id="shopping-list"></ul>
    <button onclick="addCheese()">Add cheese</button>
  </body>
[...]
```

Listing 5: D3 example – HTML

```javascript
let data = [
  'apples',
  'bananas',
  'oranges'
]

function rerender() {
  d3.select('#shopping-list')
    .selectAll("li")
    .data(data)
    .enter()
    .append("li")
    .style('color', 'red')
    .text(d => d)
    .style('opacity', 0.0)
    .transition()
    .duration(500)
    .style('opacity', 1.0)
}

function addCheese() {
  data.push('cheese')
  rerender()
}

rerender()
```

Listing 6: D3 example – JavaScript

5.1.4 d3-graphviz and DOT

d3-graphviz [31] combines the library graphviz [7] and d3.js. Graphviz is a library for generating static graphs with the help of the DOT language [5]. d3-graphviz adds animation and transition features of d3 on top. It is used for building the stack and heap visualization in JavaWiz.

```
digraph G {
  Node1 -> Node2 [label = "Edge label"]
  subgraph cluster {
    node [style = filled; color=lightblue]
    label = "Cluster"
    color = blue
    Node3 [label=<
        <table border="1" color="black">
            <tr><td>HTML</td><td port="p1">label</td></tr>
        </table>
        >]
    Node3:p1 -> Node4
  }

  Node2:s -> Node4:w
}
```

Listing 7: DOT language example

In Listing 7 and Figure 21, a small example of the DOT language and its rendering result can be seen. The essential features are the following:

- Nodes in the graph are created by writing down their name (see `Node1` and `Node2`), with optional formatting settings provided in brackets (see `Node3`).
- Edges are defined by an arrow between two node names, again with settings in brackets. An example here is the edge label between `Node1` and `Node2`.
- Edges can have ports defining where the arrow starts on a node and where it ends on another node. For example, `Node2:s -> Node4:w` means that the arrows starts on the “south” (bottom) of `Node2` and ends on the “west” (left) of `Node4`.

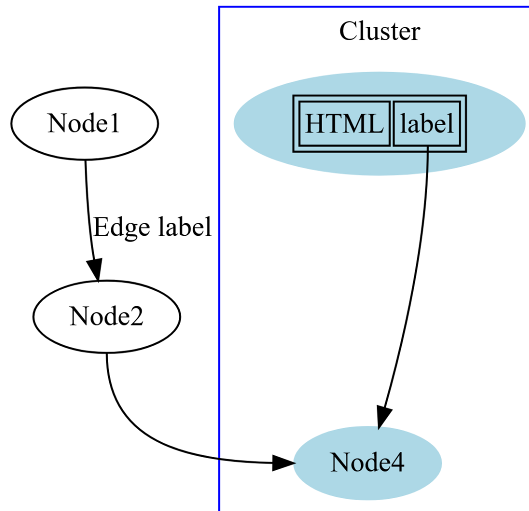


Figure 21: Rendering result of Listing 7

- A cluster is an area that may contain multiple nodes. Its formatting options, nodes and edges are put between curly brackets.
- The layouting of the nodes is done automatically, which is one of the main reasons why we chose (d3-)graphviz. Small adjustments and work-arounds are possible, for example using ports or adding invisible elements to influence the layouting result.

Another feature we heavily used are *HTML-like labels*. Instead of using a string for the label attribute, like for the edge between `Node1` and `Node2` in Listing 7, a template similar to HTML can be used to create more complex nodes (see `Node3`). It is also possible to add ports to individual elements within the label. In line 12, an edge between the port `Node3:p1` and `Node4` is defined. In the rendering result, only the table cell with the content "label" is pointing at `Node4` and not the whole `Node3`.

5.1.5 JDI

The JDI (Java Debug Interface) [18] is an API allowing developers to access debug information of a running virtual machine and control its execution. It is a central

part of the JavaWiz backend, as it is essentially a debugger built with the JDI (written in Kotlin).

5.1.6 Java Parser

The JavaParser [12] is a library that creates an Abstract Syntax Tree (AST) for Java code. This AST can then be analyzed and manipulated programmatically. In JavaWiz, it was used to inject helper structures and variables into the code written by the user. These helpers made it possible to display the condition expressions in the desk test (see Section 5.6).

5.1.7 Monaco Editor

The Monaco editor by Microsoft [17] is a customisable editor that can be used on the web. It can be customised to provide useful highlights during the program execution.

5.2 Architecture

The architecture of the system is shown in Figure 22. The system consists of a backend and a frontend. The backend is responsible for compilation and debugging. The frontend shows the code in an editor and visualizes the execution trace it receives from the backend.

The execution flow is as follows:

- When the user presses the start button, the code is sent to the backend in a `COMPILE` command.
- The backend will compile the code, start a debugger after successful compilation and make the first step.
- A click on the "step" button by the user will send a `STEP` command, causing the debugger to perform a "step into", i.e., it will also enter methods.
- Sending input from the console component will dispatch an `INPUT` command with the input value to the backend, where the debugger handles the input.

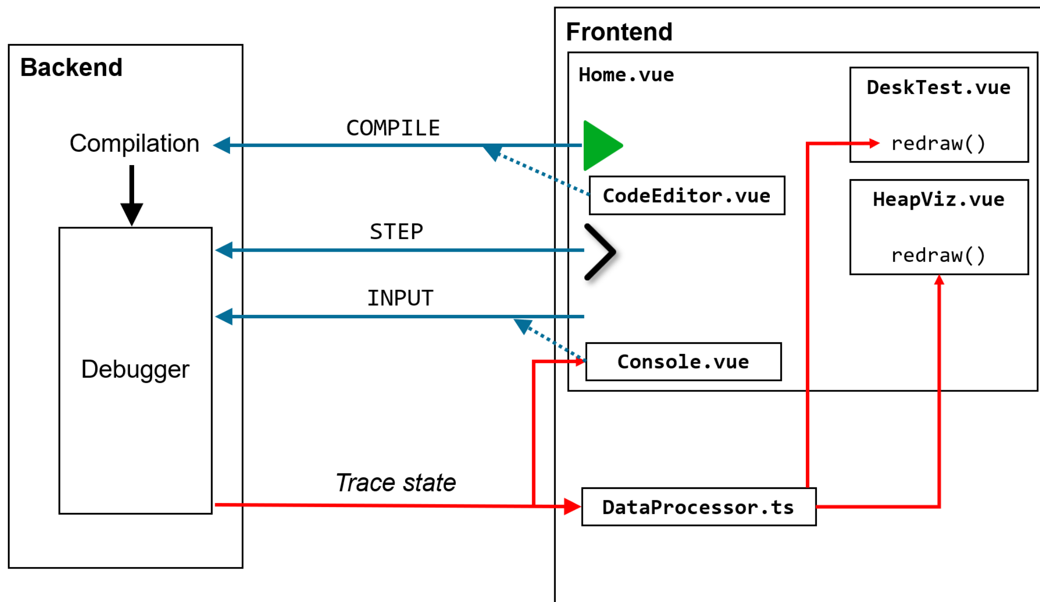


Figure 22: JavaWiz Architecture.

- As a result of these actions, a single trace state is sent back to the frontend.
- Every state the frontend receives will be appended to a list, which makes it possible to step back through previous states.
- To allow highlighting changes in the visualizations, the traces have to be compared. This is done after every step made by the user (i.e., as soon as two trace states are available for comparison) and the resulting "processed trace state" is forwarded to the visualizations, which will be redrawn.

The backend is a web socket server that handles the incoming requests. It will initiate the code compilation and start a debugger. As soon as the debugger is running, the backend is ready to handle step and input requests from the frontend, record the results of this request and send them back to the frontend. The frontend has one parent component, `Home.vue`, that contains all other components and is responsible for communicating with the backend. In the upcoming chapter, the communication between the two is described in more detail.

5.3 Communication

We start with the assumption that both frontend and backend are running.

Pressing the "start" button calls the function `startCompilation()` in `Home.vue`. This function will send a message to the backend that contains a task description and parameters. An example is shown in Listing 8, which contains the `COMPILE` message with the user code as a parameter.

```
1 vm.websocket.send(JSON.stringify({
2   task: 'COMPILE',
3   parameters: [vm.editorText]
4 }))
```

Listing 8: Compilation request.

The receiving part in the backend is the `DebugWebSocketServer.kt`. This web socket server understands three requests: `COMPILE`, `STEP`, `INPUT`. The `COMPILE` request will trigger the following actions:

1. **Modification**: The source code is modified in a certain way so it can handle displaying conditions and their evaluation results in the desk test - see Section 5.6 for more details.
2. **Compilation**: In the course "Software Development 1", students receive the classes `In.java` and `Out.java` to simplify handling input and output in the beginning. In the browser version of the tool, which can only handle a single user-defined file, these two files are compiled together with the modified source code, so they are ready to use in the tool.
3. **Response**:
   - Successful compilation: a debugger will be started, the first step will be executed and its result is sent back via the websocket to the frontend.
   - Failed compilation: the compilation output is sent back to the frontend.

When the debugger was successfully started, the frontend will display more buttons:

- Clicking the *next* button will call the function `nextTime()` in `Home.vue`. This will send a message containing the task description `STEP` and no parameters. The `DebugWebSocketServer.kt` will make a step in the debugger and send back the resulting trace state. Each successful step will also increase the `stateIndex` in `Home.vue` that indicates at which position in the trace we currently are.

- The *reset* button calls the `startCompilation()` function again, for quickly jumping back to the beginning of the program.

- The *back* button does not involve communication with the backend. It will decrease the `stateIndex` and thus move back to a previous state of the program.

If the program expects input, the user can enter something into the text field of the console component and press "Send". The function `sendInput()` will then send a message with the task `INPUT` and the parameter `inputValue` taken from the text field. In `DebugWebSocketServer.kt`, this input is then provided to the debugger and the result is sent back to the frontend.

The responses made by the backend use the interface shown in Listing 9 (`Response.kt`).

```
1 data class Response(
2     val request: Request?,
3     val status: TaskResult,
4     val data: Any?,
5     val error: String?
6 )
```

Listing 9: Response interface.

The field `request` contains the original request (task and parameters) made by the frontend. `status` indicates if the request failed or not. `data` contains the data generated by the backend, i.e. a trace state. `error` may contain a string with more information after an unsuccessful request, for example, the compile error message.

On the frontend side, `Home.vue` will create a new `WebSocket` during initialization (method `connect()` called from the `mounted()` lifecycle method) and register four event handlers on it:

- `onopen` will be called when the connection is ready and sets off a "Connection successful" notification for the user.
- `onmessage` defines how the responses from the backend are handled. There are three possibilities:
  - The latest response came after a `COMPILE` task and was successful: the frontend receives the initial trace state and assigns it to the `trace` list.
  - The latest response came after a `COMPILE` task and failed: the frontend will show a "Compilation failed" notification.
  - The latest response came after a `STEP` task and was successful: the frontend receives another trace state, concatenates it to the `trace` list and increases the `stateIndex`.
- `onclose` reacts to a closed connection (both intentional and not) by showing a notification to the user.
- `onerror` also shows a notification to the user, in case the connection failed. This will most likely happen when the debugger is not running.

At this point, we know *how* and *when* communication takes place. Now is the time to look at *what* data is actually sent to the frontend and how it is processed.

5.4 Processing the Trace

The basic trace state delivered by the backend can be seen in Listing 10.

```
1  export interface TraceState {
2    readonly kind: 'TraceState'
3    readonly sourceFile: string
4    readonly line: number // currently active line
5    readonly stack: StackFrame[] // stack
6    readonly heap: HeapItem[] // active references to Arrays,
      Strings and Objects
7    readonly loadedClasses: LoadedClass[] // classes together with
      their static fields
8    readonly output: string // stdout output produced since the last
       step
9    readonly error: string // stderr output produced since the last
      step
10   readonly input: string // stdin since the last step
11 }
```

Listing 10: Trace state interface.

Every successful response from the backend will contain one state. In the frontend, these states are concatenated to an array. This array `trace` is a local property of `Home.vue`.

One major feature of JavaWiz is showing what exactly has changed after a step. Without this, it is significantly harder to follow a program, especially for beginners. This is why the trace needs to be processed before it is passed to the components.

The processing becomes relevant after the first step by the user has been made. Then, the states can be compared. The method `processTrace(...)` in `DataProcessor.ts` first maps the currently existing trace states to a `ProcessedTraceState`. This structure essentially contains the data from the previous *and* the next trace state, i.e., there is a `heap` and a `nextHeap`, `stack` and `nextStack` and so on.

```
const currentLocalVars: LocalVar[] = state.stack.flatMap((
    stackFrame: StackFrame) => stackFrame.localVariables)
const nextLocalVars: LocalVar[] = state.nextStack.flatMap((
    stackFrame: StackFrame) => stackFrame.localVariables)

nextLocalVars.map((localVar: LocalVar) => {
  localVar.changed = false
})

const changedVars: LocalVar[] = _.differenceWith(
  nextLocalVars,
  currentLocalVars,
  (next: LocalVar, curr: LocalVar) =>
    next.name === curr.name && _.isEqual(next.value, curr.value)
  )
)

changedVars.forEach((v: LocalVar) => {
v.changed = true
})
```

Listing 11: Comparing trace states.

For each of these states, all the locals, statics and the heap contents are compared. In Listing 11, the code for comparing local variables is shown (static variables and heap objects are treated similarly):

- Each variable has a boolean flag `changed` to indicate their status. First, this flag is set to false for every variable.

- Then, they are compared with the method `differenceWith(...)` from the library "lodash" [16]. It will return the elements that are in the first array (in this example, `nextLocalVars`), but not in the second. The custom comparator function uses another lodash method called `isEqual`, which is a *deep* equal. This means that nested properties will also be compared.

- The `changed` flags of the elements in the resulting array are set to true. This flag can then be accessed by the visualizations to highlight changed elements.

Performance. This approach is pretty straightforward, but has a downside: after every step, the `changed` flags for the whole trace history are recomputed. It would only be necessary to compare the newest states, i.e. compute the changes incrementally. For really small programs, like the ones beginners usually write, the slow down was not really noticeable. Thus, improving the trace processing was not a top priority.

Desk test data. The resulting `processedTrace` is used for the stack and heap visualization. For the desk test, this trace is further processed. Essentially, it is turned into another data structure optimized for usage with d3.js. The motivation for this was to avoid non-visualization related code in the desk test component.

A special requirement for the desk test is that it shows the complete history of the program. Columns will not disappear when a method is exited, for example. Therefore, the method `computeDeskTestLines(...)` in `DataProcessor.ts` flatmaps *all* trace states and creates sets of locals, statics and conditions (see Listing 12). The lodash function `uniqWith` creates a set using `isEqual` as the comparator. These sets are later used to render the headers (see Section 5.6). For the table rows that only contain changed values, a separate array of type `DeskTestLine` is created. The elements in this array have a property `updatedVal` that will only be set if `changed` is true. The corresponding interfaces can be found in `DeskTestData.ts`.

```
1    data.currentStatics = _.uniqWith(
2      processedTrace.flatMap((state: ProcessedTraceState) => state
   .nextLoadedClasses || [])
3        .flatMap((clazz: LoadedClass) => clazz.staticFields)
4        .map((field: StaticVar) => {
5          return {
6            class: field.class,
7            name: field.name
8          }
9        }),
10     _.isEqual)
```

Listing 12: Computing desk test information for static variables.

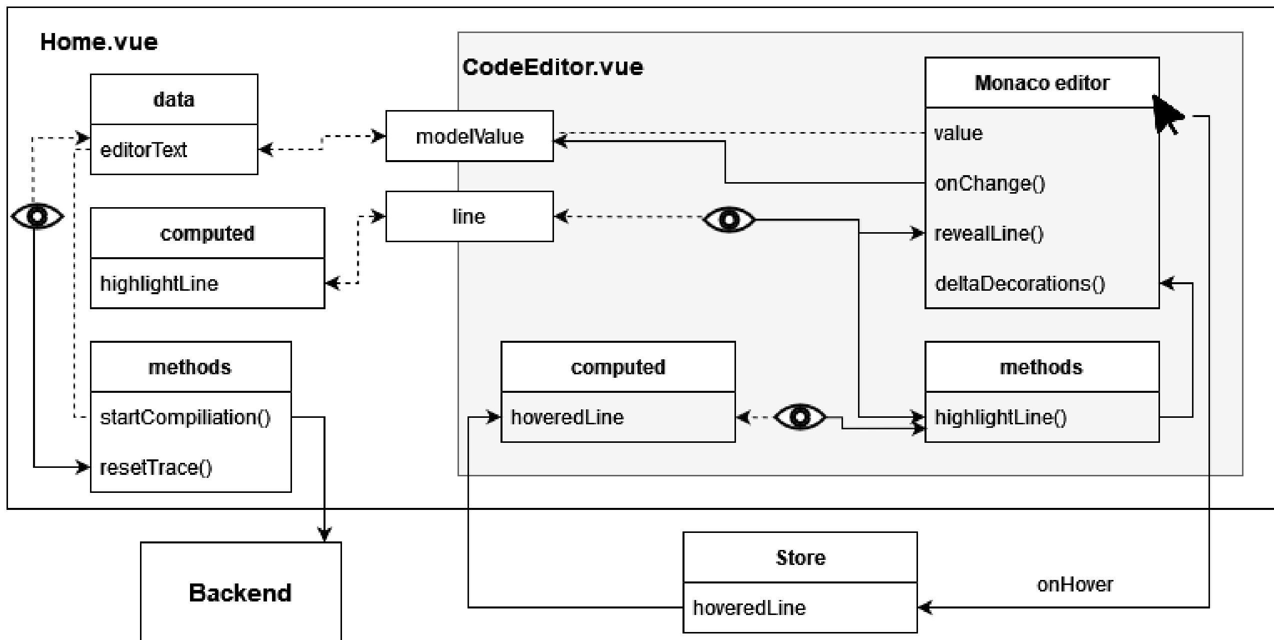


Figure 23: Editor component.

5.5 Editor

Figure 23 shows the parts and data flow of the editor component. The central part of this component is a Monaco editor instance. The following properties and methods of the Monaco editor are relevant for us:

- `value`: The current editor content.
- `onDidChangeModelContent`: An event listener that reacts to changes of the editor content (shortened to `onChange` in Figure 23).
- `revealLine`: A method that scrolls to the line it receives as a parameter.
- `deltaDecorations`: A method taking care of the line highlighting. It will transform the old decorations to the new ones with a minimum number of operations.

The editor content needs to be accessed by `Home.vue`, as this is the component responsible for sending the code to the backend. To make this work, the local

property `editorText` in `Home.vue` is passed as a property named `modelValue`. When the user types in the Monaco editor, it will trigger the `onDidChangeModelContent` listener. The function then emits a `update:modelValue` event to assign the Monaco editor `value` to the `modelValue` property. This is necessary because mutating props directly is not allowed in Vue.

The second property the editor receives is the line on which the debugger currently halted. A watcher is installed on this line property. If it changes, `editor.revealLine()` and `highlightLine()` are called. The former makes sure that the editor scrolls to the line if it is not visible currently, the latter calls the function `deltaDecorations` to set up the actual highlighting. Listing 13 shows how highlighting the next line to be executed looks like.

```
1  this.decorations = this.editor.deltaDecorations(this.decorations,
     [
2    {
3    range: new monaco.Range(this.line, 1, this.line, 1),
4    options: {
5      isWholeLine: true,
6      className: 'editor-highlighted-line',
7      zIndex: 0
8    }
9    }
10 ]
11 )
```

Listing 13: Adding a line highlighting.

The first parameter is the local field `decorations` in the editor component containing the previous decorations. The second parameter is an array with an object containing the target range and CSS class of a new decoration.

The editor is one of the components where the Vuex store is used. To sync the hovered line between editor and desk test, the hovered line number is saved to the store. In the editor component, `hoveredLine` is a computed property that retrieves the current value from the store. We also registered a watcher on this property. If it changes, the method `highlightline()` will be called.

In the `Home.vue` component, a watcher on the `editorText` triggers the method

`resetTrace()`. Changing the editor content will cause the whole application to reset. The content of `editorText` will be sent to the backend when the start/compile button is pressed.

5.6 Desk Test

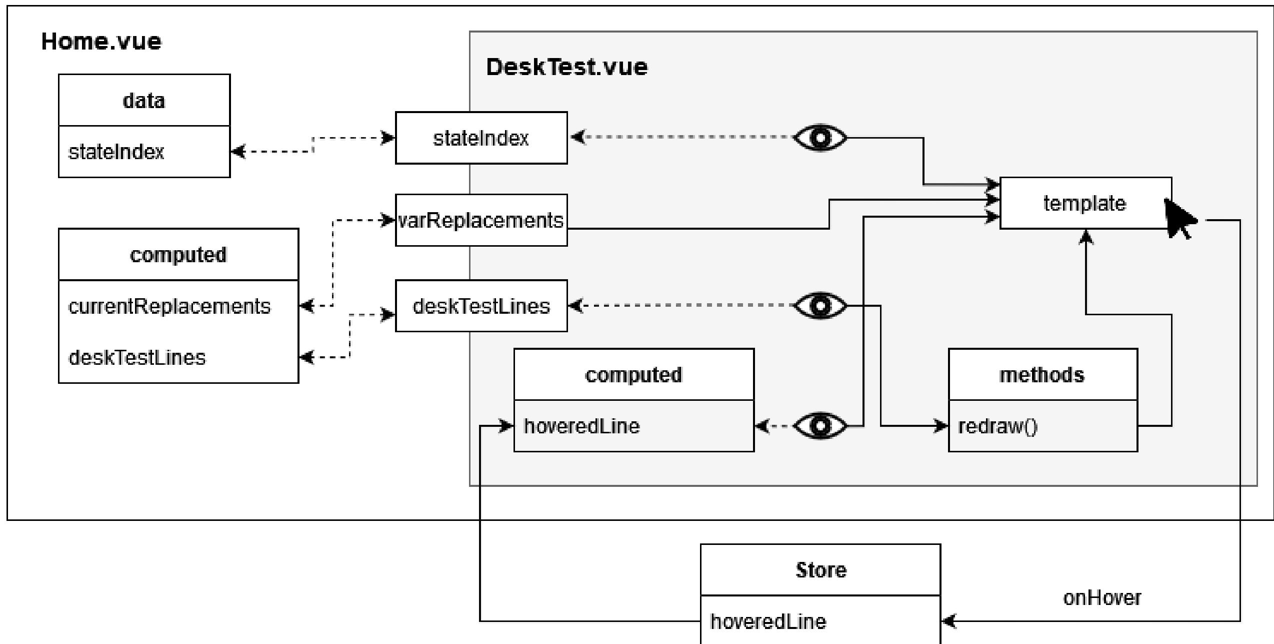


Figure 24: Desk test component.

Figure 24 shows how the desk test component works. It receives three props:

- The current state index: if it is zero (= no steps made), a text will be displayed instead of a table, informing the user about how to navigate through the program.
- The map `varReplacements`: it is required for the conditions column and is discussed in the "Conditions" paragraph at the end of this section.
- The `deskTestLines`: the actual desk test data, which is used to construct the table with d3.js.

A watcher installed on the `deskTestLines` will call the `redraw()` function on update.

In Listing 14, a part of the top header showing the "local/static/conditions" text is created. A `th` element with the text "static" is inserted. The columns have a fixed order, which is why `enter.insert()` is used rather than `enter.append()`. The data passed to d3.js is the number of statics, which is used to set the `colspan` attribute. All other parts of the header are created similarly.

```
1  const staticVarHeaderData = vm.deskTestLines.currentStatics.length
       > 0
2    ? [vm.deskTestLines.currentStatics.length]
3    : []
4
5  const varDescHeader = tableRoot.select('#var-desc-header')
6
7  varDescHeader.selectAll('.static-var-header')
8    .data(staticVarHeaderData)
9    .join(enter => enter.insert('th', '.condition-header'))
10   .classed('static-var-header truncate', true)
11   .attr('colspan', d => d)
12   .text('static')
```

Listing 14: Creating the desk test "static" header.

```
1  const tableBody = tableRoot.select('tbody')
2
3  const tableBodyRows = tableBody.selectAll('.value-row')
4    .data(vm.deskTestLines.lines)
5    .join(enter => {
6    const tr = enter.append('tr')
7      .attr('class', function (d) {
8      return 'l' + d.line
9      })
10     .classed('value-row', true)
11     .on('mouseover', (_, d) => {
12     vm.$store.commit('changeLine', { lineNr: d.line, className: vm
        .currentClass })
13     vm.$store.commit('updateHoverLocation', false)
```

```
14      })
15      .on('mouseout', () => {
16      vm.$store.commit('changeLine', { lineNr: -1, className: vm.
    currentClass })
17      })
18
19    tr.append('td')
20      .classed('line-nr', true)
21      .text(d => d.line)
22
23    tr.append('td')
24      .classed('line-cell', true)
25      .append('div')
26      .classed('line-text truncate', true)
27      .text(d => d.editorText)
28
29    common.fadeInTransition(tr.selectAll('td'), common.
    MEDIUM_TRANSITION_DURATION)
30    return tr
31    })
32
33  tableBodyRows.selectAll('.var-value-cell')
34    .data(d => d.vars)
35    .join(enter => {
36    const td = enter.insert('td', '.static-value-cell, .condition-
    value-cell')
37      .classed('var-value-cell', true)
38      .classed('value-cell', true)
39    common.fadeInTransition(td, common.MEDIUM_TRANSITION_DURATION)
40    td.each(function (d, _i) {
41      vm.appendRepresentations(d3.select(this), d)
42    })
43    return td
44    }
45    )
46    [...]
```

Listing 15: Creating the desk test rows.

In Listing 15, we take care of the table rows with the data from `desktestLines.lines`. First, the table body is selected, and within the table body, we select the rows with class `.value-row`. In the `enter` function, a `tr` element is appended and modified to enable the row highlighting (see Paragraph "Row highlighting"). Two `td` elements are appended for the first two columns – one receives the line number as text, the other the code line. A fade-in transition is added to the `td` elements (see `Common.ts`).

The rest of the table cells in a row receive the data from `deskTestLines.lines[...].vars`. Because the `tableBodyRows` are already bound to `deskTestLines.lines`, we only need to pass `d => d.vars` to the `data()` function. In the `join` function, `td` elements are inserted (again, in the right order), a fade-in transition is added, and `appendRepresentations` is called for every `td` element. This function appends text to the table cell depending on its type. For example, primitive values are added directly, and strings are put between quotes and additionally receive a CSS class for shortening them (Listing 16).

```
1  [...]
2  if (data.updatedVal.primitive) {
3    representationBox.append('div')
4      .text(data.updatedVal.primitive)
5  }
6  if (data.updatedVal.string) {
7    const textDiv = representationBox.append('div')
8    textDiv.classed('truncate', true)
9    textDiv.text('"' + data.updatedVal.string + '"')
10 }
11 [...]
```

Listing 16: Part of `appendRepresentations()`.

Row highlighting. For the row highlighting, the `tr` element needs three things:

- A class attribute with the line number,
- a `mouseover` event handler that commits the line number of the hovered row to the store, and

- a `mouseout` event handler that resets the line number in the store to `-1` when the desk test is no longer hovered.

Additionally, the desk test component has a computed property `hoveredLine` that retrieves the line number from the store. If the value in the store is changed by hovering the editor, the computed property will be updated. A watcher on this property triggers the selection of the `tr` with the corresponding line number and adds a CSS class for the highlighting (see Listing 17).

```
1  watch: {
2    hoveredLine: function () {
3      const vm = this
4      d3.selectAll('tr td:not(.overlay-cell)')
5      .classed('highlighted', false)
6
7      d3.selectAll('tr').filter('.l' + vm.hoveredLine)
8      .selectAll('td:not(.overlay-cell)')
9      .classed('highlighted', true)
10   },
11 [...]
```

Listing 17: Watcher for highlighting desk test rows.

Conditions. An important feature of the desk test is the conditions column, where the expression of a condition is displayed in the header and its (changed) evaluation result in the rows below. This required some changes in the backend. Before compiling, the code is parsed by the JavaParser. The resulting `CompilationUnit` accepts multiple *visitors*, one for each statement where a condition might occur. The general task of these visitors is to extract the condition, assign it to a "hidden" variable, replace the condition with the variable and add update statements accordingly.

In the example in Listing 18 and 19, two `_debugger_boolean` are added to the source code. The data sent to the frontend contains a map with the names of these variables and their respective condition expressions, for example `"_debugger_boolean_0" -> "i < 5"`. As mentioned before, the desk test component receives this map as a

prop. The expression string is then simply added to the condition column headers.

```java
class Test {
    public static void main(String args) {
        int i = 0;
        while(i < 5) {
            if(i % 2 == 0) {
                System.out.println(i + "is an even number");
            }
            i++;
        }
    }
}
```

Listing 18: Original source code before modifications.

```java
class Test {
    public static void main(String args) {
        int i = 0;
        boolean _debugger_boolean_1 = i < 5;
        while(_debugger_boolean_1) {
      boolean _debugger_boolean_0 = i % 2 == 0;
      if(_debugger_boolean_0) {
        System.out.println(i + "is an even number");
      }
      i++;
      _debugger_boolean_1 = i < 5;
    }
    }
}
```

Listing 19: Modified source code.

The editor will still contain the original code, while the debugger in the backend actually executes the modified source code. To make it seem like it is still executing the original code, a line map is utilized. The line map of the code examples is shown in Listing 20. If a line does not exist in the old code, the map points to `-1`. The debugger will continue stepping until it has also executed an original line before sending a trace state back.

```
New -> Old
1 -> 1
2 -> 2
3 -> 3
4 -> -1
5 -> 4
6 -> -1
[...]
```

Listing 20: Line map.

5.7 Stack and Heap

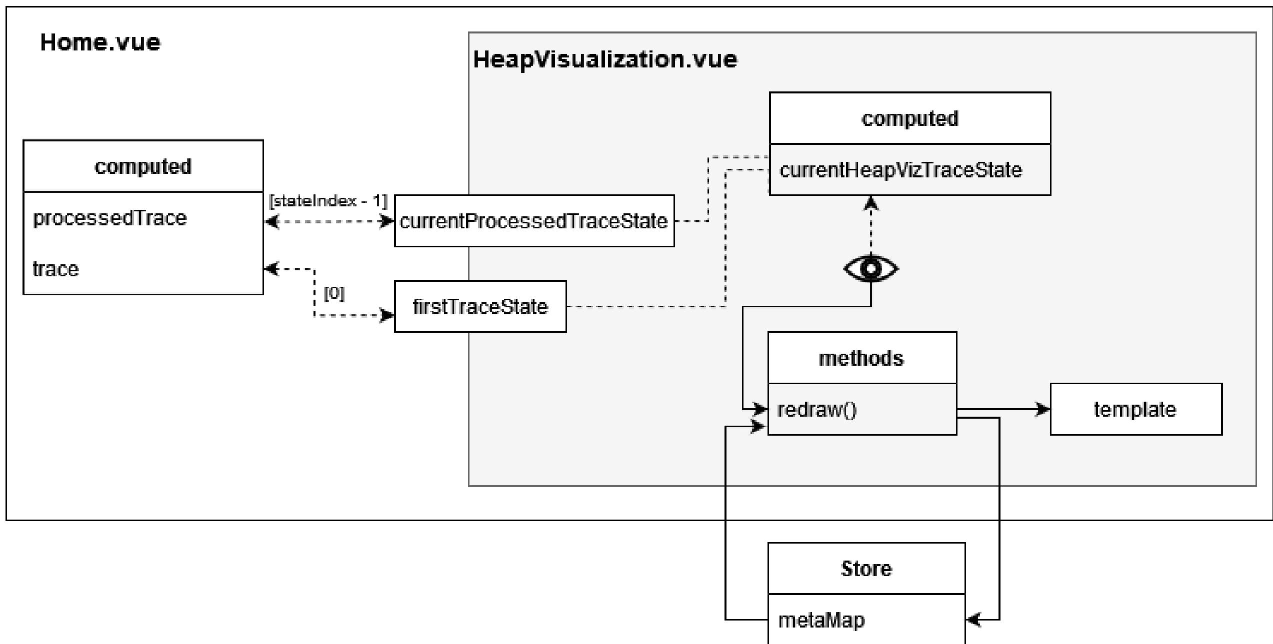


Figure 25: Stack and heap component.

Figure 25 shows the data flow in the stack and heap component. It receives two props: `currentProcessedTraceState` (`processedTrace[stateIndex - 1]`) and `firstTraceState` (`trace[0]`).

For rendering the visualization, we essentially need one big DOT string constructed from `currentProcessedTraceState`. This process is as follows:

- First, `currentProcessedTraceState` is transformed into `currentHeapVizTraceState` of type `HeapVizTraceState`. The elements in `processedTrace` are supplemented with *ports* for drawing edges later and unique *identifiers* for showing/hiding elements (see paragraph "Hiding elements"). Listing 21 shows part of this transformation as an example.

- A watcher on `currentHeapVizTraceState` calls the `redraw()` function.

- In `redraw()`, the DOT string is constructed from the trace information. To make this more maintainable, `HeapVisualization.vue` contains templates (`DOT_PARTS`) for the individual parts of the string. Some of them are static, like basic graph and table structures, and some contain placeholder items (pre- and postfixed by `###`). Listing 22 shows the template for rendering a string object as an example.

```
1  function mapStackFrame (sf: StackFrame, stackFrameNr: number):
       HeapVizStackFrame {
2    return {
3    [...]
4      localVariables: sf.localVariables.map(function (lv: Var):
       HeapVizVar {
5        return {
6          kind: 'HeapVizVar',
7          name: lv.name,
8          type: lv.type,
9          value: valToHeapVizVal(lv.value),
10         changed: lv.changed,
11         port: `l_${stackFrameNr}_${sanitizeIdentifier(lv.name)}`,
12         identifier: `roots:l_${stackFrameNr}_${sanitizeIdentifier(
       lv.name)}`
13       }
14     }),
15   [...]
16   }
17 }
```

Listing 21: Extending the processed trace state for the stack and heap component.

```
stringRow:
  `     <tr>
          <td sides="R"></td>
          <td width="${MIN_WIDTH}" colspan="3">
            <font color="###HIGHLIGHT_COLOR###">
              <###HIGHLIGHT_TAG###>"###VALUE###"</###
HIGHLIGHT_TAG###>
            </font>
          </td>
          <td sides="L"></td>
        </tr>
`,
```

Listing 22: Template for a row containing a string.

Hiding elements. An important feature of this component is the possibility to hide parts of the graph, as the heap can grow quite fast. Hidden elements should stay hidden when the user keeps stepping. Since the visualization is rerendered after every step, erasing all state, the information about hidden elements is saved in a map in the Vuex store. Before creating the DOT string from the trace, the heap is traversed in the method `calcNodeVisibility()` to determine which objects will be hidden with the help of this map.

Buttons to collapse and expand edges were added to the graph (Figure 26). The plus symbol next to reference variables is essentially a link, i.e. the `href` attribute is set for this table cell. The content of the `href` attribute is the `identifier` we added before. With the help of d3.js, this attribute can be selected (see Listing 23). The method `toggleExpandedIdentifier` then commits the value to the store.

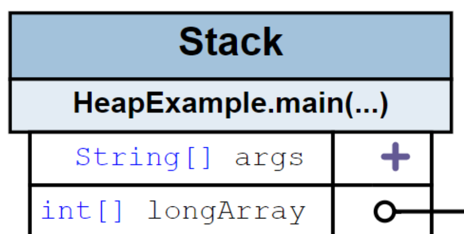


Figure 26: Collapsed and expanded variables in the stack.

```
d3.selectAll('g:not(large-heap-element) a')
  .on('click.other', function (event: any) {
  event.preventDefault() // to stop navigation

  vm.toggleExpandedIdentifier(d3.select(this).attr('href'))

  vm.redraw()
  })
```

Listing 23: Toggling the visibility of elements.

Similarly, large objects and arrays are collapsed by default. Instead of a plus symbol, a "... X more/show less" text is rendered. Figures 27 and 28 show how this looks like.

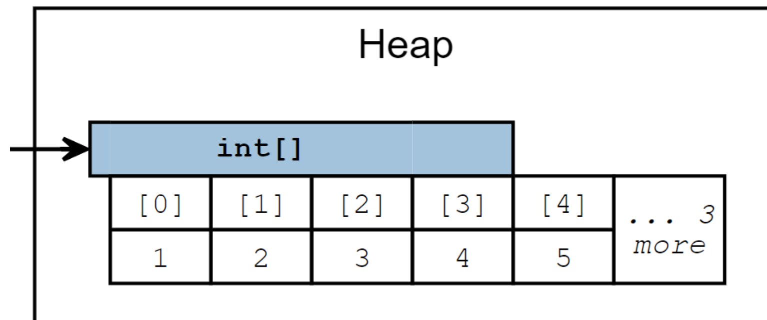


Figure 27: Collapsed large array.

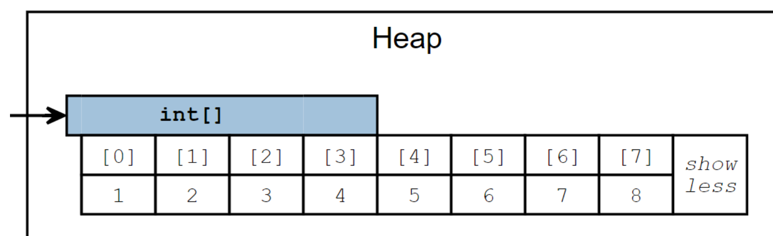


Figure 28: Expanded large array.

First trace state. The stack and heap component should already show information before the user has made a step, in contrast to the desk test where the initial visualization is empty. We might have static fields and `args` to show, for example. Because the `processedTrace` only exists from after the first step onwards, we need a "first trace state" that we can use for the initial render. This first trace state comes from the original, unprocessed `trace`. The changed flags of all local and static variables are set to true for that state.

Layouting issues. The automatic layouting reduces the coding effort tremendously, however, it has its downsides. In an earlier version of the visualization, the stack and the statics table were separate nodes. Changes in the heap sometimes caused both tables to switch places after every step, making it hard to follow. The workaround was to create a single node with a single table and use an invisible table cell to divide stack and statics. When rerendering after a step, the table contents would often also "fly" to their final position, even though they did not changed. We noticed that keeping the number of tags consistent solved this: if a text is supposed to be highlighted in red, the corresponding `<font color="...` tag is not *added* to the string, but rather always exists, just with a different `color` attribute.

5.8 Console

The console component has four main parts:

- A `div` that contains the input, output and error text of the trace states,
- A header with a "live/replay" indicator,
- An input field and
- A button to send the input.

The console receives a computed property called `slicedTrace`. As the name suggests, this property only contains part of the trace. When stepping back, past outputs and inputs should disappear. For this, it is necessary to only keep

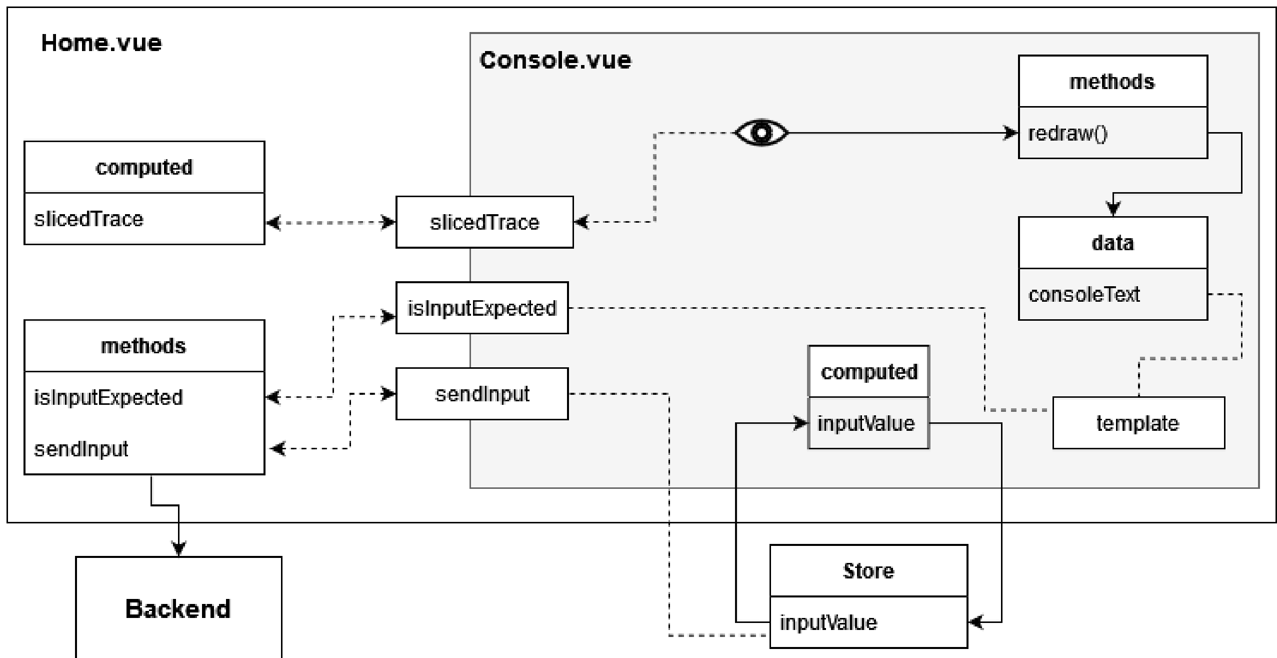


Figure 29: Console component.

the states up until the current stateIndex (inclusive). A watcher installed on `slicedTrace` calls the `redraw()` method.

In the `redraw()` method, the output, input and error contents of all states in `slicedTrace` are mapped into a single string. This string is a local property (see `data()` function) in `TheConsole.vue` and is added to the output `div` with text interpolation. The interpolated text is additionally wrapped in `pre` tags to preserve all whitespace. This ensures that all output will look exactly like in a real terminal.

The input field is a regular HTML input bound with `v-model` to the computed property `inputValue`. Usually, this only requires a local property to work. However, in our case, sending the input to the debugger takes place in the parent `Home.vue`. This means we need to get the content of `inputValue` there somehow. One way of solving this is to put the content into the Vuex store. For this to work, the computed property is defined with a getter and setter method interacting with the store.

The component receives two more properties from `Home.vue`: the methods `sendInput` and `isInputExpected`. The method `sendInput` will be triggered by click-

ing the button. The button's `disabled` property is set by the return value of `isInputExpected`, meaning that the button will only be activated if the program currently expects input. The method simply checks with a regex if the line we currently stand on uses any methods from the `In` or `Scanner` class.

5.9 Extension

The browser version of JavaWiz in its current state has one major downside: it clashes with the "ready-to-use" requirement we defined in the beginning. Students would have to clone the project and start it locally as described in Section 4.3.2, which can be overwhelming for beginners. To provide an easier setup, a Visual Studio Code extension was created by my colleague Simon Grünbacher. The extension essentially packages the debugger .jar and a frontend build and interacts with the VS Code API. Instead of the web editor and console, it uses the VS Code editor and a pseudo-terminal to simulate the console. This version has multiple advantages:

- JavaWiz can be simply installed from the VS Code Marketplace.
- Instead of jumping from the browser to a fully-fledged IDE, beginners can become accustomed to the more lightweight VS Code first. VS Code is also recommended in the MOOC and setup tutorials are provided.
- The web version is a single-file tool. With some modifications of both frontend and backend, the extension enables multi-file usage of JavaWiz.

However, it is a third major part of the project that needs to be maintained. In Section 6, we also mention potential future functionality that would be difficult to realise with the extension.

6 Conclusion and Future Work

This thesis described the visualization tool JavaWiz for supporting programming novices in executing Java programs. First, different visualization tools for programming beginners were analyzed. We found that they either did not fit the teaching approach at JKU, were outdated, or lacked crucial features. Based on those findings, the tool JavaWiz was created. It allows users to step through their Java programs line by line and visualize each statement's impact on the memory state. With the "desk test", users are enabled to observe the execution of statements and changes of variable values. The stack and heap visualization helps with understanding object-oriented programming. The tool also allows live user input – it is essentially a simplified visual debugger and does not have to execute the whole program first. We focused on making the tool as easy to use as possible.

An open question that remains is the platform on which JavaWiz will be distributed. Both the web version and the extension have advantages and disadvantages, which we briefly discussed in Section 5.9.

JavaWiz was first used in the JKU course "Software Development 1" in winter semester 2022 to collect feedback from both teachers and students. The feedback was mostly positive and provided valuable ideas for improvement. In October 2023, the MOOC will go live and JavaWiz will tap into the new user group of self-learning students.

This thesis described the basic version of JavaWiz. Between completing the practical part and writing this thesis, several features have been added already:

- Felix Schenk added two visualization components: one for visualizing linked lists and one for binary trees [35].
- Andreas Schlömicher added a flowchart component [36].
- Melissa Sen implemented an input stream visualization.

Besides new visualization components, a few other ideas came up that did not fit the scope of this thesis.

The Online Java Tutor we saw in Related Work could be embedded. Embeddable visualization components could be used to enhance online tutorials and digital textbooks. For communication between teachers and students, or among students, shareable states could be a useful feature. For example, if a student struggles with a particular execution point, a link could be sent to a peer that opens JavaWiz at the exact same point. However, both embedded and shareable visualizations would require hosting the tool somewhere for public access. With the VS Code extension, these features might not be feasible at all.

In Section 3, we saw the tool Jsvee for creating visualizations on the expression evaluation level. This could be useful for showing a more detailed execution instead of just line by line. A feature like this could be implemented hand-in-hand with the option to switch between an "easy" and "advanced" mode. As the number of features will continue to grow, there should be some way to tailor the tool for different user groups (beginners, intermediate students, teachers etc.). The contents of the follow-up course "Software Development 2" at JKU could be a useful guideline for future advanced features.

Finally, user studies can be made to professionalize and systemize the feedback pipeline.

List of Figures

1	Example of a desk test for a simple loop.	8
2	Example of a typical simple object visualization.	8
3	Example of a typical simple array visualization.	8
4	BlueJ window with class operations. Source: [3]	10
5	Greenfoot Programming Environment. Source: [33]	12
6	JAVAVIS main window. From: [13]	12
7	Java Tutor. Screenshot taken from [10].	14
8	Jeliot. Taken from [15].	16
9	Jsvee. Screenshot taken from [11].	16
10	JavaWiz overview.	19
11	JavaWiz in the Visual Studio Code Marketplace.	19
12	JavaWiz VS Code extension after starting.	20
13	Overview of the JavaWiz components.	21
14	Notification when compilation was successful.	23
15	Desk test code example.	25
16	Desk test execution result.	25
17	Example of the stack and heap visualization.	26
18	Two-dimensional array in the heap visualization.	27
19	Console example.	28
20	Rendering result of Listing 1 and Listing 2	33
21	Rendering result of Listing 7	38
22	JavaWiz Architecture.	40
23	Editor component.	47
24	Desk test component.	49
25	Stack and heap component.	55
26	Collapsed and expanded variables in the stack.	57
27	Collapsed large array.	58
28	Expanded large array.	58
29	Console component.	60

Listings

1 App.vue . . . 31
2 ChildComponent.vue . . . 32
3 Vuex store example. . . . 34
4 Vuex store usage. . . . 34
5 D3 example – HTML . . . 36
6 D3 example – JavaScript . . . 36
7 DOT language example . . . 37
8 Compilation request. . . . 41
9 Response interface. . . . 42
10 Trace state interface. . . . 44
11 Comparing trace states. . . . 45
12 Computing desk test information for static variables. . . . 46
13 Adding a line highlighting. . . . 48
14 Creating the desk test “static” header. . . . 50
15 Creating the desk test rows. . . . 50
16 Part of `appendRepresentations()`. . . . 52
17 Watcher for highlighting desk test rows. . . . 53
18 Original source code before modifications. . . . 54
19 Modified source code. . . . 54
20 Line map. . . . 55
21 Extending the processed trace state for the stack and heap component. 56
22 Template for a row containing a string. . . . 57
23 Toggling the visibility of elements. . . . 58

References

[1] Angular. https://angular.io/. Accessed: 2023-08-23.

[2] BlueJ. https://www.bluej.org/. Accessed: 2023-03-09.

[3] BlueJ Tutorial. https://www.bluej.org/tutorial/tutorial-v4.pdf. Accessed: 2023-04-27.

[4] D3 by Observable | The JavaScript library for bespoke data visualization. https://d3js.org/. Accessed: 2023-08-23.

[5] DOT Language. https://graphviz.org/doc/info/lang.html. Accessed: 2023-08-23.

[6] eInformatics@Austria. https://www.tuwien.at/einformatics/. Accessed: 2023-07-27.

[7] Graphviz. https://graphviz.org/. Accessed: 2023-03-08.

[8] Greenfoot. https://www.greenfoot.org/door. Accessed: 2023-03-09.

[9] iMooX. https://imoox.at/mooc/. Accessed: 2023-03-15.

[10] Java debugger - Java Tutor - Learn Java programming by visualizing code. https://pythontutor.com/java.html#mode=edit. Accessed: 2023-05-01.

[11] Java Example with a loop. https://acos.cs.aalto.fi/html/jsvee/jsvee-java/ae_JavaTutorial_4_7_8. Accessed: 2023-07-06.

[12] JavaParser - Home. https://javaparser.org/. Accessed: 2023-08-23.

[13] JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI) | SpringerLink. doi:10.1007/3-540-45875-1.

[14] JavaWiz. https://github.com/SSW-JKU/javawiz. Accessed: 2023-08-28.

[15] Jeliot :: Home. https://cs.joensuu.fi/jeliot/. Accessed: 2023-05-02.

[16] Lodash. https://lodash.com/. Accessed: 2023-08-23.

[17] Monaco Editor. https://microsoft.github.io/monaco-editor/. Accessed: 2023-03-08.

[18] Overview (Java Debug Interface). https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/index.html. Accessed: 2023-08-23.

[19] Python Tutor unsupported features. https://docs.google.com/document/d/13_Bc-l2FKMgwPx4dZb0sv7eMfYMHhRVgBRShha8kgbU/edit?usp=embed_facebook. Accessed: 2023-05-01.

[20] React. https://react.dev/. Accessed: 2023-08-23.

[21] Scratch - Imagine, Program, Share. https://scratch.mit.edu/. Accessed: 2022-11-21.

[22] Visual Studio Code - Code Editing. Redefined. https://code.visualstudio.com/. Accessed: 2023-08-28.

[23] Visual Studio Marketplace. https://marketplace.visualstudio.com/vscode. Accessed: 2023-08-28.

[24] Vue.js - The Progressive JavaScript Framework | Vue.js. https://vuejs.org/. Accessed: 2023-03-08.

[25] What is Vuex? | Vuex. https://vuex.vuejs.org/. Accessed: 2023-08-23.

[26] M. Ben-Ari, R. Bednarik, R. Ben-Bassat Levy, G. Ebel, A. Moreno, N. Myller, and E. Sutinen. A decade of research and development on program animation: The Jeliot experience. *Journal of Visual Languages & Computing*, 22(5):375–384, Oct. 2011. doi:10.1016/j.jvlc.2011.04.004.

[27] J. Boustedt, A. Eckerdal, and R. McCartney. Threshold concepts in computer science.

[28] D. Cornell. Mastery Learning: 10 Examples, Strengths & Limitations. https://helpfulprofessor.com/mastery-learning/. Accessed: 2023-04-27.

[29] B. Du Boulay. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2(1):57–73, Feb. 1986. doi:10.2190/3LFX-9RRF-67T8-UVK9.

[30] P. J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, Mar. 2013. Association for Computing Machinery. doi:10.1145/2445196.2445368.

[31] M. Jacobsson. D3-graphviz. `https://github.com/magjac/d3-graphviz`. Accessed: 2023-03-08.

[32] T. Kohn and D. Komm. Teaching Programming and Algorithmic Complexity with Tangible Machines. In S. N. Pozdniakov and V. Dagienė, editors, *Informatics in Schools. Fundamentals of Computer Science and Software Engineering*, Lecture Notes in Computer Science, pages 68–83, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-02750-6_6.

[33] M. Kölling. The Greenfoot Programming Environment. *TOCE*, 10:14, Nov. 2010. doi:10.1145/1868358.1868361.

[34] J. H. F. Meyer and R. Land. Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning. *Higher Education*, 49(3):373–388, Apr. 2005. doi:10.1007/s10734-004-6779-5.

[35] F. Schenk. *A Component for Visualizing Java Methods with Linked Lists and Binary Trees*. Bachelor's thesis, JKU, SSW, 2022.

[36] A. Schlömicher. *Visual Studio Code Plugin for Visualizing Java Methods as Flowcharts*. Bachelor's thesis, JKU, SSW, 2023, [to appear].

[37] T. Sirkiä. Jsvee & Kelmu: Creating and Tailoring Program Animations for Computing Education. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 36–45, Oct. 2016. doi:10.1109/VISSOFT.2016.24.