JavaWiz: A Trace-Based Graphical Debugger for Software Development Education

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

Simon Grünbacher
Institute for System Software
Johannes Kepler University
Linz, Austria
simon.gruenbacher@jku.at

Herbert Prähofer
Institute for System Software
Johannes Kepler University
Linz, Austria
herbert.praehofer@jku.at

***Abstract*—Software development education faces challenges in teaching abstract and complex programming concepts. Since problems in comprehension can lead to decreased student engagement, we introduce JavaWiz: an educational graphical debugger that addresses these challenges by combining traditional debugging functionality with intuitive, dynamic visualizations of program state and run-time behavior.**

JavaWiz's key features include real-time visualization of heap, stack, and static fields; automatically generated flow charts; interactive representations of data structures; and unique time-travel debugging capabilities. Its step-by-step visual exploration of code execution, including the ability to step backward, bridges the gap between abstract concepts and concrete program understanding.

We present the tool's visualization components in detail and discuss its applications in teaching. Lecturers report positive influence on their in-class demonstrations and initial student feedback reinforces the tool's usefulness for program comprehension.

***Index Terms*—Software Education, Program Understanding, Graphical Debugger, Control Flow, Heap, Stack, Data Structures**

I. Introduction

Software development education presents unique challenges stemming from the abstract syntax and semantics of programming concepts [1], [2]. Students often struggle to comprehend a program's dynamic behavior, leading to difficulties in understanding core principles such as memory management, data structures, and control flow [3]–[5]. Working with arrays, recursion, and error handling can also be particularly challenging due to the cognitive load they impose on beginners. This can result in decreased comprehension and engagement, negatively influencing learning outcomes [6].

Effective visualizations have been shown to significantly improve learning outcomes in computer science [7]. Provided with visual representations of abstract concepts, students can build better mental models of program behavior [8], improve their problem-solving skills, and increase their overall engagement with the material [9], [10]. This aligns with cognitive load theory, which suggests that properly designed visual representations can facilitate more efficient learning [11].

While traditional debuggers are powerful tools for experienced developers, they often prove overwhelming and hard to learn for beginners [12]–[15]. These tools typically present a wealth of information without the necessary context or visual cues that novice programmers require, potentially discouraging students in their early stages of learning.

To address these challenges we present *JavaWiz*, an innovative graphical debugger [16] specifically designed for software development education. It combines features of a traditional debugger with intuitive, dynamic visualizations of program state and execution flow. Uniquely, JavaWiz offers "time-travel" capabilities, allowing users to step backward through program execution and re-inspect past operations [17], [18].

JavaWiz leverages Abstract Syntax Tree (AST) extraction to analyze source code and the Java Debug Interface (JDI) to perform debug steps [19], [20]. This technical foundation allows JavaWiz to provide real-time memory visualizations as well as automatically generated flow charts and interactive representations of various data structures. JavaWiz keeps track of all changes to fields, variables, I/O, conditions, and more at every debug step, enabling comprehensive state tracking and "time-travel" navigation. Its configurable user interface allows educators and students to tailor the tool to their specific needs and current learning objectives. To ensure widespread accessibility and ease of use, JavaWiz[1] is available as open-source project[2] and as a VS Code extension[3]. This way, students can install JavaWiz with a single mouse click directly within their development environment. Additional information, together with a video of the tool, can be found at https://javawiz.net/.

Summarized, our contributions encompass:

- An architectural overview of JavaWiz (Section II)
- Details on the data it collects (Section III)
- A comprehensive set of educational visualizations:
  - Automatically generated flow charts (Section IV-A)
  - Detailed memory visualization (Section IV-B)
  - Tabular view of variable changes (Section IV-C)
  - Animated visualizations of linked lists and binary trees in the program (Section IV-D) as well as 1D and 2D array operations (Section IV-E)
  - Input buffer monitoring (Section IV-F)
- A discussion on how JavaWiz can be used to supplement software development courses and preliminary feedback on its utility and usefulness (Section V).
- an outlook for possible future studies (Section VIII).

[1] https://javawiz.net/ (including video of the tool)
[2] https://github.com/SSW-JKU/javawiz
[3] https://marketplace.visualstudio.com/items?itemName=SSW-JKU.javawiz

This is the author's version of the work. The definitive version will be published in the Proceedings of the 33rd IEEE/ACM International Conference on Program Comprehension (ICPC 2025).

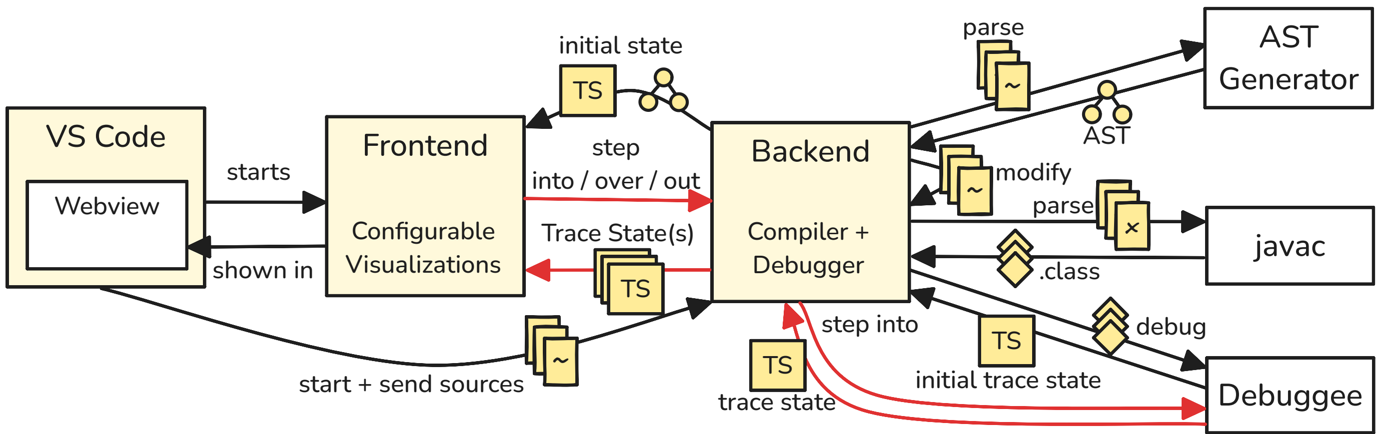

Fig. 1: (1) The frontend depicts the program state in different visualizations. (2) The backend extracts ASTs, instruments and compiles the user code, and runs the compile result as a debuggee. From there, it collects all data needed for visualization on debug steps. (3) The VS Code extension bundles all components and displays JavaWiz's frontend in a web view component.

II. Architecture

This section outlines JavaWiz's architecture, focusing on the system's interconnected components, its user interface and possible user interactions.

A. Components

In Figure 1, the system's main interconnected components are shown. They work together to facilitate the visual debugging of Java programs. Black operations are performed when starting JavaWiz, red operations are performed when the user steps through the program. Communication between Visual Studio Code, the frontend and the backend is performed via WebSockets, exchanging data in JSON format.

1) Visual Studio Code (VS Code) Extension: JavaWiz is easily accessible through a VS Code extension, streamlining the installation process for users. Once installed, the extension integrates a "Run in JavaWiz" feature directly into the code editor, available via codelens and the context menu (see Figure 2). On click, the extension initiates the JavaWiz backend and frontend. The backend instruments and compiles the user's code and serves as the debugger, while the frontend presents visualizations and is hosted in a web view within VS Code.

2) Backend: Upon receiving the source code (which can consist of multiple files) it first extracts an Abstract Syntax Tree (AST) for each class and performs code instrumentation to extract additional information from the user's program for visualization. Then, the backend compiles the modified

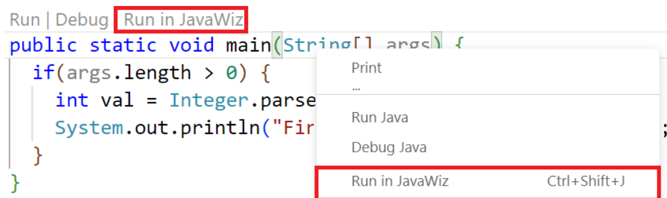

Fig. 2: Various ways to start JavaWiz from within VS Code.

Java files and switches into the role as debugger: it launches the program as debuggee, steps into its `main` method and generates the initial trace state which is then, together with the AST, transmitted to the frontend. The backend then handles various step events, such as step over or step out, requested by the frontend, by internally executing multiple step into operations, ensuring comprehensive trace state generation.

3) Frontend: The frontend, a web application, offers a customizable interface to select and display various visualizations, such as flowcharts, memory visualizations, array operations and more based on the user's needs. Users can configure the number and layout of visualizations and can visually inspect their program while stepping through code execution. Users can also replay past operations. When stepping back in time, past program steps are replayed from the execution trace, while stepping in live mode triggers backend requests for real-time trace state updates.

B. User Interface and User Interactions

On its initial launch, JavaWiz presents a simple interface with a flow chart and a memory visualization, see Figure 3. This minimizes information overload while offering essential insights into program execution, improving the tool's *learnability* [21]. Users can personalize the UI, with the option to display one to four resizable visualizations simultaneously, see Figure 4. JavaWiz remembers the user's last setup, restoring the same configuration upon subsequent launches.

The user interface includes a toolbar at the top of the screen, facilitating program navigation through buttons or keyboard shortcuts. Features include stepping into, over, out, and running to a specified line, as well as restarting the debugger to clear the current execution trace. Designed with responsiveness in mind, JavaWiz adapts its layout to accommodate various screen sizes and resolutions. For instance, step buttons stack vertically when horizontal space is limited, ensuring usability even when zoomed in during teaching sessions on projectors, as shown in Figure 5.

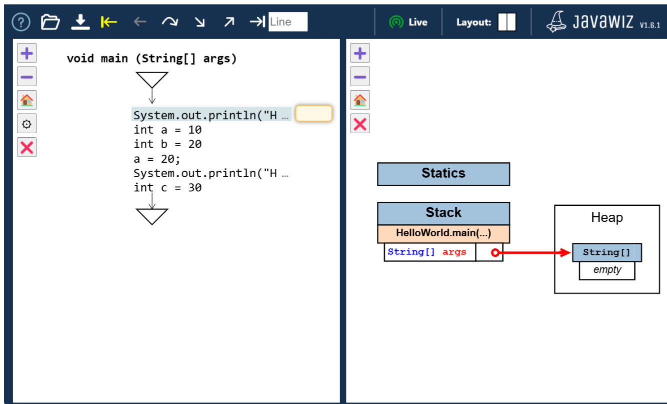

Fig. 3: On its initial launch, JavaWiz only shows two of its visualizations to ease the learning-by-doing onboarding process [22], [23].

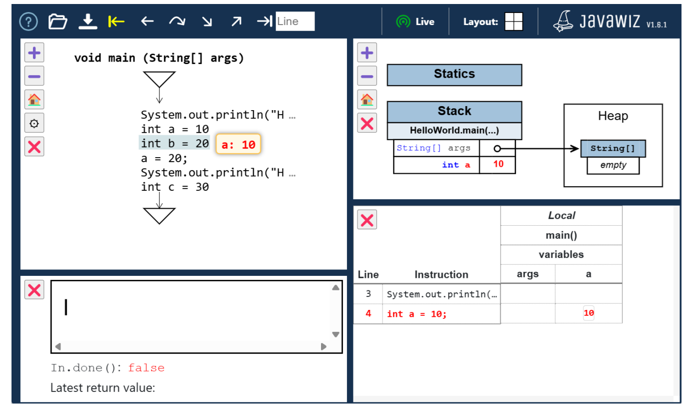

Fig. 4: Lecturers can tailor the view to current teaching needs, while students can adjust their views based on certain comprehension problems encountered in homeworks.

III. Data

Before presenting JavaWiz's visualizations, we outline the key data sources and methods to collect them. This includes forming an execution trace that contains trace states that encapsulate the program state, e.g., variables and their values, the value of conditions or I/O operations at a certain point in the program execution. JavaWiz also utilizes abstract syntax trees (ASTs) for program flow visualization. Understanding these fundamental data sources is crucial for comprehending based on which data JavaWiz creates its educational visualizations.

A. Variables and Values

In programming, a *variable* is essentially a named container for data (i.e., *value*). It thus consists of two key components:

- **Name:** Identifier used to reference the variable in code.
- **Value:** Data stored in the variable, which is either:
  - *Primitive:* Basic data types such as integers, floating-point numbers, or boolean values.
  - *Reference:* Pointer to a heap object, including arrays.
  - `null`*:* Value indicating the absence of a reference.

1) Variable Collection: To provide comprehensive visualizations, JavaWiz collects information about all variables and their values from three main sources:

- **Loaded Classes:** JavaWiz scans all loaded classes to collect their *static fields* and their values.
- **Stack Trace:** JavaWiz examines all active methods in the stack trace to gather all *local variables* and their values.
- **Heap:** Starting from static and local variables, JavaWiz traverses the heap to collect:
  - all *objects* including their respective *fields*.
  - all *arrays* including all their *elements*.

  Objects and arrays are given a unique ID in this process which stays consistent throughout the whole execution.

To optimize performance and maintain clarity in our visualizations, we deliberately exclude certain objects and methods from tracing. For example, we do not track the (static) fields of standard library objects, e.g., we do not track the the internals of an `java.util.ArrayList`, nor do we step into method calls to the Java standard library. This decision is driven by two primary considerations: First, to avoid overwhelming users with potentially irrelevant and confusing information about system internals. Second, to mitigate the performance challenges associated with tracing, storing, and displaying a vast number of objects.

2) Variable Identification: As shown in Figure 6, JavaWiz uses a systematic naming convention to identify each variable.

- **Static variables** `s_<fqclassname>_<fieldname>`
  - Example: `s_a.b.Reader_EOF` for static field `EOF` in class `Reader` in package `a.b`
- **Local variables** `l_<depth>_<varname>` – The same method (and thus variable) can occur multiple times in the stack. We thus use the methods' stack frame depth in the identifier, starting with depth 0 at the `main` method.
  - Example: `l_0_x` for local variable `x` in `main`
- **Object fields** `f_o<objectid>_<fieldname>`
  - Example: `f_o40_age` for field `age` in object 40
- **Array elements** `a_o<objectid>_<index>`
  - Example: `a_o30_2` for index 2 in array 30

This identification scheme allows JavaWiz to track each occurrence of every variable, even in complex scenarios involving recursion or multiple instances of the same class.

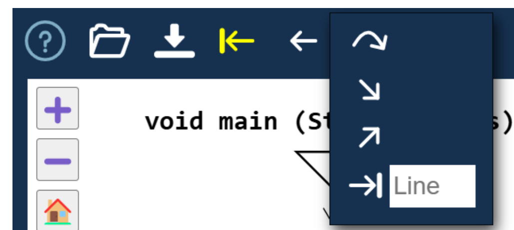

Fig. 5: Stacked step buttons ("step over", "step into", "step out", "run to line") on small screens.

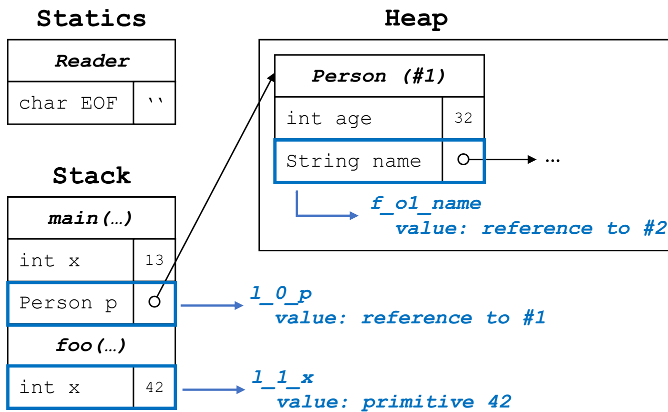

Fig. 6: Identifiers enable us to uniquely identify variables. Reference values enable us to follow points-to relations.

B. Conditionals and Array Accesses

Visualizing run-time information is crucial for bridging the gap between abstract code and tangible program behavior, thereby enhancing the educational experience for learners. Yet, certain visualizations in JavaWiz require information about run-time data that are not directly available in standard debugging APIs such as JDI, namely the evaluation of *conditions* and accesses to *arrays*. This section presents how JavaWiz effectively captures this information to power its educational visualizations.

1) Conditionals: Tracing the evaluation of conditional expressions poses unique challenges since re-evaluating them can lead to unintended side effects. For example, consider the expression `calcAndPrint() < 100`, where `calcAndPrint` performs a calculation and outputs a message. Re-executing this condition, for example by using a tracing technique as shown in Listing 1, would result in duplicate outputs, i.e. unintended side effects. Similar, problems would arise if the function involved non-deterministic elements such as `Math.random()`.

```
// Trace condition value, executing calcAndPrint()
record(calcAndPrint() > 10);
// Original code also calls calcAndPrint()!
// Performs a second print (unintended side effect)
while(calcAndPrint() > 10) { ... }
```

Listing 1: Careless tracing can lead to unintended side effects.

To address this, JavaWiz employs a custom method `recordCondition` similar to Listing 2. The debugger registers calls to this function and stores the condition value in the current trace state.

```
boolean recordCondition(String conditionName,
                        boolean conditionValue) {
  // Here the debugger hooks in to collect
  // the condition into the trace state
  return conditionValue;
}
```

Listing 2: Simplified implementation of the `recordCondition` tracing function.

Calls to this utility method are added for all conditions in all control flow structures in the user code, as shown in Listing 3.

```
// Original
while(calcAndPrint() < 100) { ... }

// Instrumented
while(recordCondition("calcAndPrint() < 100",
                      calcAndPrint() < 100)) { ... }
```

Listing 3: Condition instrumentation example.

This approach ensures important properties:

- The expression is evaluated only once, preserving its side effects.
- No additional source lines are introduced, maintaining the integrity of exception stack traces.

2) Array Accesses: Array accesses are handled in a similar fashion. Here, we particularly have to make sure that multi-dimensional array accesses are correctly recorded. JavaWiz employs an instrumentation method `recordAccess` similar to Listing 4 to record array access details.

```
int recordAccess(Object[] arr, int idx,
                 String idxExpr, int dim) {
  // Here the debugger hooks in to collect
  // the array access info into the trace state
  return idx;
}
```

Listing 4: Simplified implementation of the `recordAccess` tracing function.

The debugger uses `recordAccess` to capture

- which array is accessed (`arr`)
- at which index (`idx`)
- using which expression (`idxExpr`)
- at which dimension (`dim`)

Calls to this utility method are added for all array accesses in the user code, as shown in Listing 5. This detailed capture ensures the ability to precisely visualize array operations, including even combined expressions such as `j + 1`, enabling students to easier understand complex data manipulations.

```
// Original
int tmp = arr[5][j + 1];

// Instrumented
int tmp = a[recordAccess(a, 5, "5", 1)]
           [recordAccess(a, j + 1, "j + 1", 0)];
```

Listing 5: Array access instrumentation example.

C. Execution Trace

The execution trace is the most fundamental component of JavaWiz, capturing the program state at each step. This section outlines how JavaWiz constructs and utilizes execution traces to provide insightful visualizations.

1) Trace Collection: JavaWiz operates by compiling and executing the user's program in a debug mode using the Java Debug Interface (JDI). This enables JavaWiz to register interest in certain events [24] such as "STEP_INTO" to be able to react to every program step. Upon each step, JavaWiz collects a *trace state*, which includes:

- **Current Location**: Identifies the file, class, method, and line number currently being executed.
- **Call Stack**: Captures the current function call hierarchy, including local variables, conditionals and array accesses.
- **Heap Data**: Includes arrays, strings, and object instances.
- **Static Fields**: Monitors all loaded classes and their static fields.
- **I/O Activity**: Records text written to `stdout` and `stderr`, as well as `stdin` input since the last step.
- **Input Buffer Content**: Additionally to read text from `stdin` we also peek at the input buffer's remaining data.

Each collected trace state is stored in a `TraceState[]` array on the frontend, forming a comprehensive execution trace. This trace enables JavaWiz's unique *time-travel* debugging feature, allowing users to step backward and visually review past program states and operations. This capability is particularly useful in educational contexts, where instructors may need to revisit specific steps during live demonstrations upon student request or to explain certain program parts in more detail.

2) Scalability Considerations: While designed primarily for educational use, JavaWiz accommodates larger programs by limiting the trace history to a specified number of steps (e.g., 1000 steps). This ensures a certain heap memory usage limit without sacrificing essential details necessary for understanding program behavior.

3) Language Agnosticism: Even though stating Java in its name, JavaWiz's architecture is inherently language-agnostic. Potential adaptation to other strongly-typed languages could be introduced in the future, provided one builds a debugging environment that can provide JSON data corresponding to JavaWiz trace states schema (see above).

D. Abstract Syntax Tree

JavaWiz primarily uses the abstract syntax tree (AST) for generating flow chart visualizations. An AST is a hierarchical tree-like representation of a program's source code. Each node represents a code construct (such as control flow structures down to expressions). Figure 7 depicts a simplified AST for a `while` with a block body that contains two statements.

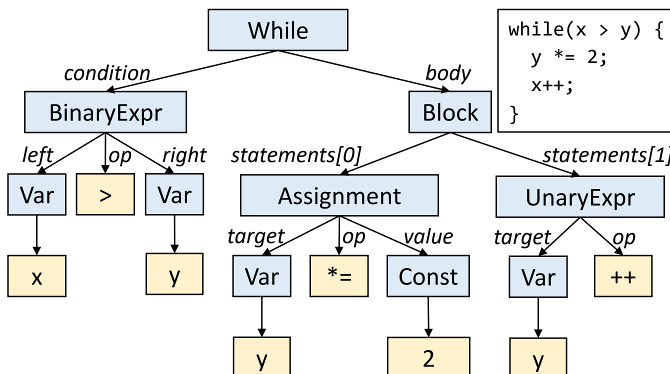

Fig. 7: The AST enables top-down program structure analysis, driving visualizations such as our Flowchart View.

The AST is a crucial component for program analysis and visualization. It can be generated using tools such as JavaParser [25] or the Java compiler (`javac`) itself.

IV. Core Visualization Components

In this section, we introduce the different visualizations that can be turned on and off by the user in JavaWiz.

A. Flowchart View

The Flowchart View presents the program using a flowchart notation [26], [27], offering a clear two-dimensional representation of the program's control flow. While primarily visualizing the static program structure, it dynamically highlights the currently active statement and displays current variable values alongside it during debugging sessions. Additionally, users can collapse and expand elements and inline method calls.

Figure 8 shows the flowchart of a sample Nim game program. The flowchart depicts the program's control structure, including the method's start and return, a do-while loop, an if-statement, and an inlined `computeMove` method. The yellow box displays the value of the local variable `coins`.

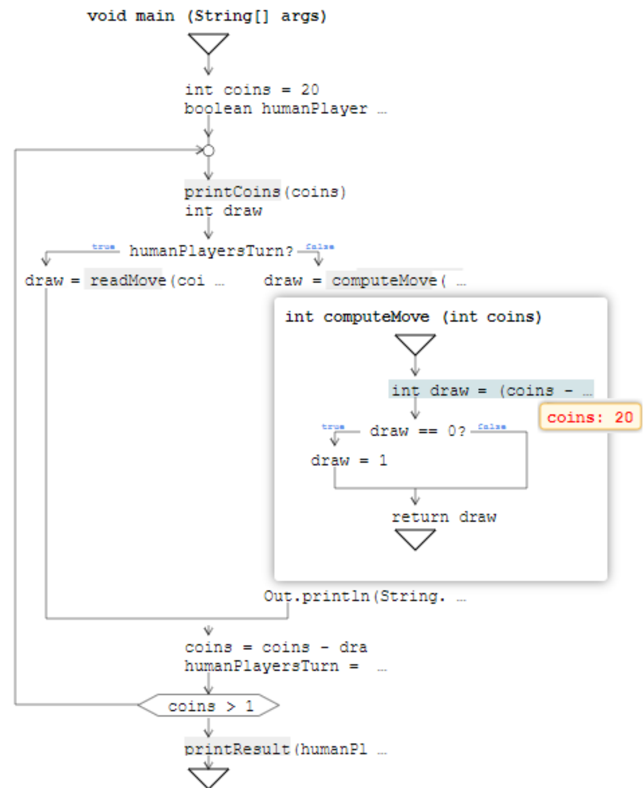

Fig. 8: Flowchart View of a program for playing the game Nim, highlighting the currently active statement in blue and depicting local variable values in a yellow box next to it.

Our flowchart notation supports all control statements found in Java. We have developed specific visual representations for switch and try/catch/finally statements, as these constructs lack standardized flowchart conventions. Figure 9 illustrates the key elements of our flowchart notation.

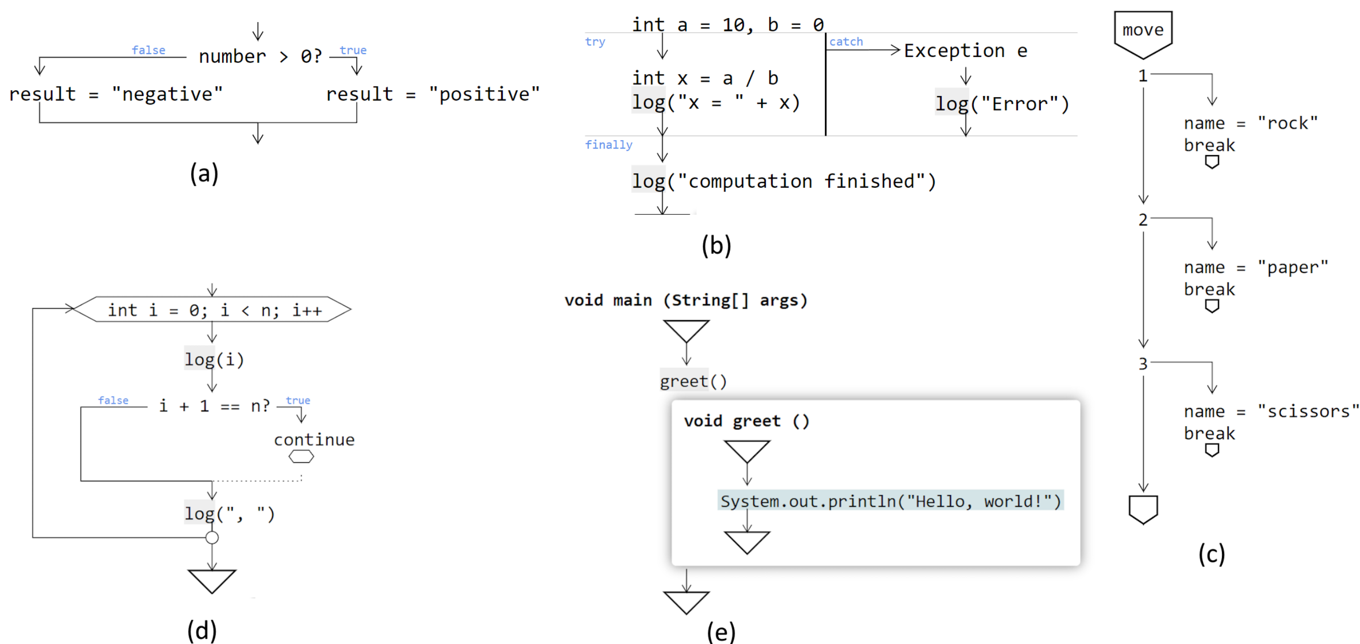

Fig. 9: Flowchart notations: (a) If statement, (b) try/catch/finally statement, (c) switch statement, (d) for loop, (e) method call.

a) *If statements* (see Figure 9(a)): The true and false branches are positioned to the right and left of the condition, labeled with `true` and `false` respectively. Which branch is shown on the left is user-configurable.

b) *Try / catch / finally statements* (see Figure 9(b)): The try/catch/finally statement is divided into three parts: The try block is shown on the left, the catch blocks are to the right (multiple catch blocks are positioned side-by-side), and the finally block is positioned below the try and catch block. The three parts are separated by fine lines.

c) *Switch statements* (see Figure 9(c)): A switch statement begins with a arrow-shaped box containing the expression or variable to be evaluated. The end of the switch is marked by a similar box without a label. Cases are listed below the start box, each labeled with its associated case value(s). The corresponding code is positioned to the right. The arrow below points to the next case. `break` statements are indicated by a small box with same shape. The notation can also depict scenarios where a case is not terminated with a `break` statement.

d) *Loops* (see Figure 9(d)): Loop headers are enclosed in a six-sided figure, and the end of the loop body is marked by a circle. For while and for loops, the box is at the top and the circle at the bottom. For do-while loops, the order is reversed (cf. Figure 8).

e) *Method calls* (see Figure 9(e)): To visualize method calls, a separate flowchart for the called method is embedded below the call expression. This chart is visible when the active statement is within the method call. The chart can also be expanded manually by clicking on it. In this case, if a call can resolve to multiple methods due to dynamic binding, the user must choose which one to expand.

Block-escaping statements: To visualize statements that exit the current block (like `break` or `return`), we have introduced statement-specific symbols placed below these statements. Table I lists the block-escaping statements and their corresponding symbols. To provide a visual cue for the control flow's destination, the symbols resemble the control structures that will be reached. For instance, the symbol for a break statement within a loop matches the symbol indicating the end of a loop. Unlike traditional flowchart notation, we do not draw an arrow from the block-escaping statement to the next statement to be executed. This approach maintains the flowchart's hierarchical structure and prevents arrows from crossing each other or statements.

break ⛉	break ○	continue ⬭	return 1 ▽	throw e ➘

TABLE I: Block-escaping statements: `break` in switch, `break` in loop, `continue` in loop, `return` from method, and `throw` exception.

B. Memory View

This visualization provides a graphical representation of a program's memory state, including its stack, static variables, and heap (see Figure 10). Its aim is to give users a comprehensive overview of the program's data and to aid users in answering questions regarding memory [28]. In particular, it's designed to aid understanding of reference semantics, particularly when multiple references point to the same object.

The visualization consists of three sections: (1) an area for the static variables, (2) the stack of method frames with local variables, and (3) the heap. Each variable is shown

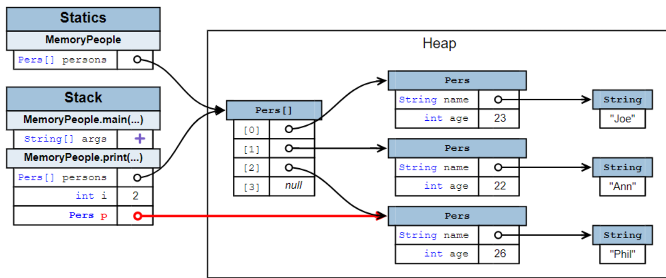

Fig. 10: Memory View example: A static variable `persons` points to an array of person objects; a local variable `persons` references the same array; local variable `p` is highlighted, indicating its recent assignment to the third object in the array.

		Local				
		main()				
		variables				conditions
Line	Instruction	a	fact	n	i	(i <= n)
3	`int fact = 1;`		1			
4	`int n = 3;`			3		
5	`int i = 1;`				1	
6	`while(i <= n) {`					true
7	`fact = fact * i;`					
8	`++i;`				2	
6	`while(i <= n) {`					true
7	`fact = fact * i;`		2			

Fig. 11: Tabular View

with its type, name, and value (primitive, reference, or null). References are depicted as arrows. Heap objects are displayed in a tabular format, with fields and their values, and with the type of the object as heading. Primitive values are directly shown next to the field's name, a reference to another object is visualized as a link to the corresponding object's node, and null is visualized with the label `null`. Arrays, strings, and exceptions have specialized representations. Arrays are displayed in a table with one row for indices and one row for elements (displayed in the same way as object fields). For strings, only the contents are displayed and internal fields of the `String` object are not shown. Similarly, for exceptions only the error message and type are displayed and object fields are hidden.

Users can expand or collapse references between objects. However, the stack frames and classes are always visible. Heap objects are only shown if they are reachable from the stack or static variables. Large arrays are initially truncated to their first couple of elements but can be expanded.

When the user steps through the program, changes to values are highlighted in red, and changes to references are indicated by highlighted links.

C. The Tabular View

The aim of the Tabular View is to help the user understand basic program execution that only manipulate primitive values. In particular, changes to variable assignment and conditions are visualized. The view is made up of a table that shows executed statements together with the values of stack variables through the program history. The conditions of loops and if-statements are also displayed. Figure 11 shows an example of this view for a simple program that computes a factorial via a loop.

D. List and Tree View

Lists and trees are fundamental dynamic data structures, but their implementation can be challenging for beginners due to the complexity of reasoning about the program state. While the Memory View presented in Section IV-B can visualize these structures, it often provides excessive detail and lacks a layout optimized for this specific task [29].

To address these shortcomings, JavaWiz offers dedicated List and Tree Views for visualizing lists and trees (see Figure 12 and Figure 13). These visualizations employ a custom layout algorithm optimized to smoothly animate common operations such as insertions and deletions. Additionally, they display local node variables alongside the nodes they reference. This makes it easier to visualize algorithms that traverse these dynamic data structures, as traversal is then just the animated movement of such pointers.

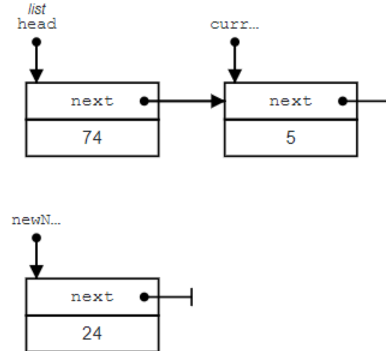

Fig. 12: List View

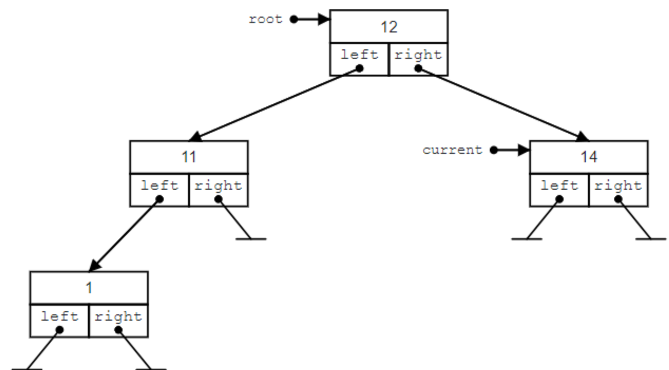

Fig. 13: Tree View

E. Array View

The aim of the Array View is to help the user understand how data is moved into and out of arrays. While the Tabular View allows for a primitive visualization of data flow and the Memory View shows the contents of an array as they change, neither can fully capture the flow of data within assignments.

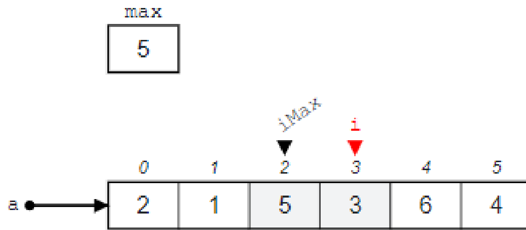

Fig. 14: Array View

To overcome this problem, the Array View shows arrays as one- or two-dimensional tables and animates index expressions that are used to access arrays. It also animates assignments involving array elements as well as temporary variables used in calculations. Figure 14 shows an example of the Array View during the execution of a method finding the index of the greatest element in an array `a`. The index `i` points to the element currently investigated, `iMax` is the index of the current maximum, and in variable `max` the current maximum value is stored.

F. Input View

Programming beginners often struggle to grasp the impact of input operations and keeping track of the input buffer's state. For example, one common pitfall is forgetting to consume the newline characters after reading an input value.

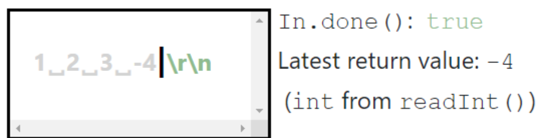

Fig. 15: Input View

To simplify input handling, students at JKU are advised to use a specific `In.java` class. This class provides static methods for reading various data types, making input operations easier for beginners. The Input View leverages a modified version of `In.java` to visualize the input buffer's current state. This helps programmers verify if the buffer contents match their expectations. Figure 15 shows an example of the Input View for a program that reads a sequence of space-separated integers. The view displays a box separating consumed and unconsumed parts of the buffer. Additionally, it shows the latest function's return value and a `done` flag indicating if the last operation was successful.

V. Usage and Feedback

Since 2022, JavaWiz has been integrated into the "Software Development 1" (CS1) course at Johannes Kepler University Linz, Austria. Annually, the course serves approximately 350 students from various technical studies (such as computer science, electronic engineering, and mechatronics) as an introduction to object-orient programming in their first semester of study. The tool has been employed both as a teaching aid for demonstrating sample programs in class as well as an aid for students in doing their homework assignments.

A. Application in Teaching

The visualization components detailed in Section IV are tailored to enhance the content of the course lectures. Table II provides a mapping between the key lecture topics and the corresponding visualization components. The course starts with a discussion of fundamental statements as well as input and output operations. The Tabular View and Input View are employed in this context. Notably, the Input View has proven invaluable in helping beginners grasp the impact of not processing all input at once and understanding what remains in the input buffer, a concept that is found difficult by students. The Flowchart View is mainly used in teaching control structures in subsequent lectures. For showing how arrays are constructed and processed, the Array View and the Memory View are used. Especially, understanding heap allocations and reference semantics have turned out to be a first real challenge in learning Java programming. In a subsequent lecture, methods and method calls are introduced. Method invocations are visualized using the Flowchart View with inlined method calls. This has proven particularly effective in conveying the concept of recursive method calls. Also the stack in the Memory View was used to help comprehending how method invocations work. The subsequent lectures focus on classes and objects. Primarily, the Memory View is employed for visualizing memory allocations and object structures. A concluding lecture introduces dynamic data structures, supported by the specific List and Tree Views.

Nr	**Lecture topic**	**Visualization component**
1	Statements and input/output	Tabular View, Input View
2	Branches	Flowchart View
3	Loops	Flowchart View
4	Arrays	Array View, Memory View
5	Methods	Flowchart View (with inlining), Memory View (stack)
6	Recursion	Flowchart View (with inlining), Memory View (stack)
7	Objects 1	Memory View
8	Objects 2	Memory View
9	Dynamic data structures	List View, Tree View

TABLE II: Table of Lectures linked to the used visualization components in class.

B. Feedback

Student feedback on JavaWiz was gathered through anonymous online questionnaires at the end of the courses. The surveys assessed students' perceptions of:

- The tool's effectiveness in classroom demonstrations
- Tool usage for homework assignments
- The tool's utility in completing homework tasks

The evaluations were conducted for both the overall tool and its individual visualization components. Further, we ask students about their prior experience in Java programming. Students were asked to respond the questions in a scale from (0) not useful, (1) little useful (2) neutral (3) useful to (4) very useful. Furthermore, the questionnaire allowed for free text responses for feedback and suggestions.

In summary, the results for classroom demonstrations were as follows:

- About 85 percent found the tool very useful or useful for demonstrations in class.
- Results are slightly better for students who had no or little prior experience in programming.
- Among the different visualizations, students found the Memory View most useful and the Tabular View least useful for classroom demonstrations.

The results for homework assignments were as follows:

- About 65 percent used JavaWiz for doing their homework.
- This number was a little higher (about 70 percent) for students who had no or only little prior experience in programming.
- About 70 percent found the tool very useful or useful for doing the homework.
- Again, results were slightly better for students who had no or little prior experience in programming.
- From the different views, students found the List View most useful for their homework assignments.

Students' free text responses suggested several improvements, many of which have been implemented over the past months and years. In particular, the evaluation after the first semester JavaWiz was used at Johannes Kepler University Linz contained suggestions for a number of useful features, such as support for multiple source files and breakpoints. In its first version, JavaWiz only supported a single source file (a drawback many other related tools, especially online tools, share) and only had a *step into* and *step back* feature. JavaWiz now supports multiple source files (including classes in mulitple packages), and JavaWiz's forward stepping features have been extended to support *step into*, *step over*, *step out* as well as a *run to line* feature similar to traditional breakpoints.

Here are some of the positive statements that the students made in their free text responses:

- "I thought JavaWiz was really cool in class because you could see very clearly what was happening in the background."
- "I found JavaWiz very good, it visualised very well how references work."
- "I am a complete beginner and JavaWiz has helped me a lot. The topics are no longer so abstract."
- "Should be a MUST for every lesson. It is very helpful!"

Further, we asked the 6 lecturers, how helpful JavaWiz was to them in classroom demonstrations. The results are as follows:

- All lecturers found the tool very useful for teaching.
- All lecturers found the Memory View and List View very useful.
- The Tabular View has been found least useful.

Also teachers were asked to give free text responses. A feedback from a teacher was "One can no longer imagine teaching stack, heap and linked lists without JavaWiz.".

VI. Limitations and Threats to Validity

While JavaWiz has shown promise in enhancing program comprehension, several limitations and threats to the validity of our study should be acknowledged:

- *Limited Evaluation Scope*: Our evaluation primarily focused on initial feedback from students and educators within a single institution. This limited scope may not fully capture the tool's effectiveness across diverse educational settings and varying levels of programming expertise.
- *User Interaction Data*: We did not collect detailed interaction data on how students use JavaWiz during their study sessions. This lack of granular usage data limits our understanding of the specific features that contribute most to learning outcomes.
- *Self-Reported Feedback*: The feedback collected was self-reported, which may introduce bias. Students and educators who had a positive experience with JavaWiz might be more inclined to participate in the survey, skewing the results.
- *Generalizability*: The feedback was collected in the context of a Java programming course at a university. The findings may not be directly applicable to other contexts (different programming languages, other education level, etc.) without further investigation.

VII. Related Work

Numerous systems and research endeavors focus on visualizing Java programs [30], [31]. For instance, Weninger et al. [18], [32]–[34] employ diverse visualization techniques, such as memory cities, to provide comprehension support for heap structures, aiding in understanding heap allocations and the identification of memory leaks. Given the extensive scope of program visualizations for Java, this section will concentrate on educational tools that effectively convey fundamental programming concepts through visualizations. Sorva et al. [10] gives a very comprehensive tool overview for programming education, also containing several tools for Java.

Many educational tools concentrate on specific aspects of a program. A prevalent focus lies in visualizing memory structures, such as the stack and heap, during Java program

execution. For instance, JAVAVIS [24], a desktop application, offers object and sequence diagrams, pioneering memory visualization for Java. Jeliot [7], [35]–[37] provides similar functionalities as a more modern alternative. EZVisor [38] is a plugin for the Netbeans IDE visualizing program executions as abstract diagrams. jGRASP [39] is an IDE that features conceptual visualization of data with object viewers. Other examples of tools for visualizing memory of Java program executions as object structures are Java Visualizer [40] or Online Java Tutor [41]. These tools are comparable to the Memory View of the JavaWiz tool in terms of functionality.

Only a limited number of visualization tools support additional aspects of Java program execution beyond memory structures. For instance, JAVAVIS [24] incorporates sequence diagram visualization. The Java Flow Visualizer [42] presents a timeline of executed statements alongside variable values, resembling a simplified version of our Tabular View and Flowchart View. Stritzinger's [43] early work on flowchart diagrams for Modula 2 programs, while employing a similar representation as our Flowchart View, lacks stepping and execution capabilities.

Debug Visualizer [44] appears to be the closest match to our tool. Similar to JavaWiz, it is also available as a VS Code extension. Moreover, it offers various visualizations comparable to ours, including graph views for heap structures, linked lists, and arrays, as well as an AST view for code structure. Its extensibility with custom views is another similarity. However, JavaWiz differentiates itself with a strict focus on Java and visualizations tailored for beginners, while Debug Visualizer provides general-purpose visualizations for multiple languages. This specialized approach offers a clear advantage in aiding novice programmers comprehending program behavior.

VIII. Future Work

While JavaWiz has shown promising results in enhancing the educational experience in Java programming, there are several avenues for future exploration and enhancement.

A. Development of New Views

To further enhance JavaWiz's educational capabilities, we envision the integration of several new views:

a) Sequence Diagram Visualization: An ongoing project is the implementation of automatically generated and evolving sequence diagrams to visualize object interactions over time. Studying these visualization, students can gain experience in object-oriented analysis and design in different contexts and domains, as recommended in related work [45]. Sequence diagrams help to provide students with a more comprehensive understanding of method calls and object lifecycles, particularly in complex systems. Such visualizations can bridge the gap between static code analysis and dynamic execution understanding. By incorporating animations, the sequence diagrams will not only enhance conceptual clarity but also serve as attractive and engaging educational resources for building teaching material.

b) LLM-driven "AI Tutor" View: As Price et al. state, *students working on programming homework do not receive the same level of support as in the classroom [46].* Thus, another potential addition to JavaWiz is inspired by advances in large language models (LLMs). This view will offer students contextual suggestions on improving their code. It will provide advice on various categories such as correctness, readability, coding style, and performance. By harnessing the power of AI, this feature will serve as a virtual tutor, offering personalized feedback and learning recommendations.

These enhancements aim to broaden the pedagogical scope of JavaWiz, making it an even more beneficial tool for software education.

B. Potential for Varying Future Studies

The introduction of JavaWiz opens a multitude of avenues for rigorous future studies. Such studies can leverage JavaWiz as a platform to conduct comparative analyses with established tools and teaching techniques, providing empirical data on its efficacy relative to traditional and contemporary educational aids.

A systematic examination of student-teacher interactions facilitated by JavaWiz can yield insights into pedagogical dynamics and the tool's role in enhancing instructional delivery.

Moreover, the impact of JavaWiz's visualizations on the comprehension of Java programming concepts and paradigms warrants a thorough investigation. This includes exploring how different learner demographics engage with the tool and the extent to which it aids in demystifying complex programming constructs. Another interesting aspect is the comparative study of graphical versus traditional debuggers. While anecdotal evidence suggests that novices may benefit from the more intuitive graphical interface that JavaWiz offers, comprehensive studies are needed to validate these claims and to understand the nuances of how such interfaces provide support.

To date, the feedback on JavaWiz has been very positive, as detailed in Section V. However, to build on this preliminary acclaim, it is imperative to conduct structured user studies that can pinpoint the specific tasks and scenarios where JavaWiz excels. These studies should aim to quantify improvements in task performance, deepen comprehension, and enhance user satisfaction. Metrics such as error rates, time to task completion, and qualitative assessments of user experience can provide a robust framework for evaluating the tool's effectiveness. Usability might be assessed using a cognitive walkthrough [47], [48] against some standard, such as Nielsen's usability attributes [49] or the *Cognitive Dimensions of Notations Framework* [50]–[53].

We aim not only to affirm the strengths of JavaWiz but also to identify areas for refinement. The ultimate goal is to evolve JavaWiz into a tool that not only resonates with students' learning styles but also contributes to a measurable decrease in course failure rates [54]–[56]. The insights garnered from such studies could inform the development of best practices for integrating JavaWiz into the curriculum, thereby enriching the educational landscape for aspiring software developers.

IX. Conclusions

This paper introduced JavaWiz, a trace-based graphical debugger specifically designed to enhance the program comprehension in Java software education. By providing an array of intuitive and interactive visualization components, JavaWiz bridges the gap between abstract programming concepts and tangible understanding. Its unique features, such as time-travel debugging and dynamic visual representations, have shown to significantly aid in the comprehension of complex programming constructs, addressing common challenges faced by novice programmers.

The deployment of JavaWiz in the "Software Development 1" course at Johannes Kepler University Linz has demonstrated its effectiveness as both a teaching and learning tool. Feedback collected from students and educators indicates a high level of satisfaction, with a majority finding the tool useful for understanding key programming concepts. The visualization components, particularly the Memory View, were highlighted as valuable assets in both classroom demonstrations and individual student assignments.

Moving forward, we aim to extend JavaWiz by integrating additional visualization features and conducting comprehensive user studies to further refine its educational impact.

In summary, JavaWiz represents a significant advancement in educational tools for software development, offering a robust and easy-to-access platform for students to engage with Java programming in a more interactive and meaningful way.

Acknowledgment

We thank the students Katrin Kern, Felix Schenk, Andreas Schlömicher, and Melissa Sen for their significant contributions to the development of JavaWiz through their master's and bachelor's thesis projects.

References

[1] D. Fonte, D. C. da Cruz, A. L. Gançarski, and P. R. Henriques, "A Flexible Dynamic System for Automatic Grading of Programming Exercises," in *Symposium on Languages, Applications and Technologies (SLATE)*, vol. 29, 2013, pp. 129–144. [Online]. Available: https://doi.org/10.4230/OASIcs.SLATE.2013.129

[2] Z. Ullah, A. Lajis, M. Jamjoom, A. H. Altalhi, A. A. Al-Ghamdi, and F. Saleem, "The effect of automatic assessment on novice programming: Strengths and limitations of existing systems," *Comput. Appl. Eng. Educ.*, vol. 26, no. 6, pp. 2328–2341, 2018. [Online]. Available: https://doi.org/10.1002/cae.21974

[3] A. V. Robins, J. Rountree, and N. Rountree, "Learning and Teaching Programming: A Review and Discussion," *Comput. Sci. Educ.*, vol. 13, no. 2, pp. 137–172, 2003. [Online]. Available: https://doi.org/10.1076/csed.13.2.137.14200

[4] E. Lahtinen, K. Ala-Mutka, and H. Järvinen, "A study of the difficulties of novice programmers," in *Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, 2005, pp. 14–18. [Online]. Available: https://doi.org/10.1145/1067445.1067453

[5] Y. Qian and J. Lehman, "Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review," *ACM Trans. Comput. Educ.*, vol. 18, no. 1, pp. 1:1–1:24, 2017. [Online]. Available: https://doi.org/10.1145/3077618

[6] T. Cui and J. Wang, "Empowering active learning: A social annotation tool for improving student engagement," *Br. J. Educ. Technol.*, vol. 55, no. 2, pp. 712–730, 2024. [Online]. Available: https://doi.org/10.1111/bjet.13403

[7] S. M. Cisar, R. Pinter, and D. Radosav, "Effectiveness of Program Visualization in Learning Java: a Case Study with Jeliot 3," *Int. J. Comput. Commun. Control*, vol. 6, no. 4, pp. 668–680, 2011. [Online]. Available: https://doi.org/10.15837/ijccc.2011.4.2094

[8] L. Ma, J. D. Ferguson, M. Roper, I. Ross, and M. Wood, "Improving the mental models held by novice programmers using cognitive conflict and jeliot visualisations," in *Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, 2009, pp. 166–170. [Online]. Available: https://doi.org/10.1145/1562877.1562931

[9] T. L. Naps, G. Rößling, V. L. Almstrum, W. P. Dann, R. Fleischer, C. D. Hundhausen, A. Korhonen, L. Malmi, M. F. McNally, S. H. Rodger, and J. Á. Velázquez-Iturbide, "Exploring the role of visualization and engagement in computer science education," *ACM SIGCSE Bull.*, vol. 35, no. 2, pp. 131–152, 2003. [Online]. Available: https://doi.org/10.1145/782941.782998

[10] J. Sorva, V. Karavirta, and L. Malmi, "A Review of Generic Program Visualization Systems for Introductory Programming Education," *ACM Trans. Comput. Educ.*, vol. 13, no. 4, pp. 15:1–15:64, 2013. [Online]. Available: https://doi.org/10.1145/2490822

[11] J. Sweller, J. J. Van Merrienboer, and F. G. Paas, "Cognitive Architecture and Instructional Design," *Educational Psychology Review*, vol. 10, no. 3, pp. 251–296, 1998. [Online]. Available: https://link.springer.com/article/10.1023/A:1022193728205

[12] R. Chmiel and M. C. Loui, "An integrated approach to instruction in debugging computer programs," in *Annual Frontiers in Education (FIE)*, vol. 3, 2003, pp. S4C–1.

[13] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: a review of the literature from an educational perspective," *Comput. Sci. Educ.*, vol. 18, no. 2, pp. 67–92, 2008. [Online]. Available: https://doi.org/10.1080/08993400802114581

[14] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers," *Comput. Sci. Educ.*, vol. 18, no. 2, pp. 93–116, 2008. [Online]. Available: https://doi.org/10.1080/08993400802114508

[15] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, "Studying the advancement in debugging practice of professional software developers," *Softw. Qual. J.*, vol. 25, no. 1, pp. 83–110, 2017. [Online]. Available: https://doi.org/10.1007/s11219-015-9294-2

[16] A. Zeller and D. Lütkehaus, "DDD - A free graphical front-end for UNIX debuggers," *ACM SIGPLAN Notices*, vol. 31, no. 1, pp. 22–27, 1996. [Online]. Available: https://doi.org/10.1145/249094.249108

[17] G. Pothier and É. Tanter, "Back to the Future: Omniscient Debugging," *IEEE Softw.*, vol. 26, no. 6, pp. 78–85, 2009. [Online]. Available: https://doi.org/10.1109/MS.2009.169

[18] M. Weninger, L. Makor, and H. Mössenböck, "Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations," in *Smart Tools and Apps in Computer Graphics (STAG)*, 2020, pp. 63–75. [Online]. Available: https://doi.org/10.2312/stag.20201241

[19] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2006. [Online]. Available: https://www.pearson.com/us/higher-education/program/Aho-Compilers-Principles-Techniques-and-Tools-2nd-Edition/PGM167067.html

[20] Oracle, "Java Debug Interface (JDI)," https://docs.oracle.com/en/java/javase/16/docs/api/jdk.jdi/com/sun/jdi/package-summary.html, 2021, accessed: 2023-08-05.

[21] T. Grossman, G. W. Fitzmaurice, and R. Attar, "A survey of software learnability: metrics, methodologies and guidelines," in *International Conference on Human Factors in Computing Systems (CHI)*, 2009, pp. 649–658. [Online]. Available: https://doi.org/10.1145/1518701.1518803

[22] J. Renz, T. Staubitz, J. Pollak, and C. Meinel, "Improving the Onboarding User Experience in MOOCs," in *International Conference on Education and New Learning Technologies (EDULEARN)*, 7-9 July, 2014 2014, pp. 3931–3941.

[23] M. Weninger, E. Gander, and H. Mössenböck, "Guided Exploration: A Method for Guiding Novice Users in Interactive Memory Monitoring Tools," *Proc. ACM Hum. Comput. Interact.*, vol. 5, no. EICS, pp. 209:1–209:34, 2021. [Online]. Available: https://doi.org/10.1145/3461731

[24] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)," in *Software Visualization, International Seminar Dagstuhl Castle*,

ser. Lecture Notes in Computer Science, vol. 2269, 2001, pp. 176–190. [Online]. Available: https://doi.org/10.1007/3-540-45875-1_14

[25] "Javaparser," https://javaparser.org/, retrieved: 2025-02-05.

[26] I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *ACM Sigplan Notices*, vol. 8, no. 8, pp. 12–26, 1973.

[27] C. Böhm and G. Jacopini, "Flow diagrams, turing machines and languages with only two formation rules," *Communications of the ACM*, vol. 9, no. 5, pp. 366–371, 1966.

[28] A. Fernandez-Blanco, A. Q. Córdova, A. Bergel, and J. P. S. Alcocer, "Asking and Answering Questions During Memory Profiling," *IEEE Trans. Software Eng.*, vol. 50, no. 5, pp. 1096–1117, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3377127

[29] T. D. Hendrix, J. H. C. II, and L. A. Barowski, "An extensible framework for providing dynamic data structure visualizations in a lightweight IDE," in *Technical Symposium on Computer Science Education (SIGCSE)*, 2004, pp. 387–391. [Online]. Available: https://doi.org/10.1145/971300.971433

[30] S. P. Reiss, "Visualizing Java in Action," in *ACM Symposium on Software Visualization (SOFTVIS)*, 2003, pp. 57–65. [Online]. Available: https://doi.org/10.1145/774833.774842

[31] A. Savidis and N. Koutsopoulos, "Interactive Object Graphs for Debuggers with Improved Visualization, Inspection and Configuration Features," in *International Symposium on Advances in Visual Computing (ISVC)*, ser. Lecture Notes in Computer Science, vol. 6938, 2011, pp. 259–268. [Online]. Available: https://doi.org/10.1007/978-3-642-24028-7_24

[32] M. Weninger, L. Makor, and H. Mössenböck, "Memory Leak Visualization using Evolving Software Cities," in *Symposium on Software Performance (SSP)*, 2019.

[33] M. Weninger, L. Makor, and H. Mössenböck, "Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor," in *Working Conference on Software Visualization (VISSOFT)*, 2020, pp. 110–121. [Online]. Available: https://doi.org/10.1109/VISSOFT51673.2020.00017

[34] M. Weninger, L. Makor, and H. Mössenböck, "Heap Evolution Analysis Using Tree Visualizations," in *Symposium on Software Performance (SSP)*, 2020.

[35] M. Ben-Ari, N. Myller, E. Sutinen, and J. Tarhio, "Perspectives on Program Animation with Jeliot," in *Software Visualization, International Seminar Dagstuhl Castle*, ser. Lecture Notes in Computer Science, vol. 2269, 2001, pp. 31–45. [Online]. Available: https://doi.org/10.1007/3-540-45875-1_3

[36] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, "Visualizing programs with Jeliot 3," in *Working Conference on Advanced Visual Interfaces (AVI)*, ser. AVI '04, 2004, p. 373–376. [Online]. Available: https://doi.org/10.1145/989863.989928

[37] M. Ben-Ari, R. Bednarik, R. B. Levy, G. Ebel, A. Moreno, N. Myller, and E. Sutinen, "A decade of research and development on program animation: The Jeliot experience," *J. Vis. Lang. Comput.*, vol. 22, no. 5, pp. 375–384, 2011. [Online]. Available: https://doi.org/10.1016/j.jvlc.2011.04.004

[38] J. Moons and C. De Backer, "The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism," *Comput. Educ.*, vol. 60, no. 1, p. 368–384, jan 2013. [Online]. Available: https://doi.org/10.1016/j.compedu.2012.08.009

[39] J. Cross, D. Hendrix, and L. Barowski, "Combining Dynamic Program Viewing and Testing in Early Computing Courses," in *International Computer Software and Applications Conference (COMPSAC)*, 08 2011, pp. 184 – 192.

[40] "Java visualizer," https://cscircles.cemc.uwaterloo.ca/java_visualize, retrieved: 2025-02-05.

[41] "Java tutor," https://pythontutor.com/java.html, retrieved: 2025-02-05.

[42] D. Pritchard and W. Gwozdz, "Java Program Flow Visualizer," https://www.cs.virginia.edu/%7Elat7h/cs1/JavaVis.html, 2025-02-05.

[43] A. Stritzinger, "Zeichnen von Ablaufdiagrammen für Modula-2-Programme," Master's thesis, Johannes Kepler University Linz, 1985.

[44] H. Dieterichs, "Debug visualizer," https://marketplace.visualstudio.com/items?itemName=hediet.debugvisualizer, 2024.

[45] V. Y. Sien, "An investigation of difficulties experienced by students developing unified modelling language (UML) class and sequence diagrams," *Comput. Sci. Educ.*, vol. 21, no. 4, pp. 317–342, 2011. [Online]. Available: https://doi.org/10.1080/08993408.2011.630127

[46] T. W. Price, J. J. Williams, J. Solyst, and S. Marwan, "Engaging Students with Instructor Solutions in Online Programming Homework," in *ACM Conference on Human Factors in Computing Systems (CHI)*, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3313831.3376857

[47] A. Bisante, V. S. V. Datla, E. Panizzi, G. Trasciatti, and S. Zeppieri, "Enhancing Interface Design with AI: An Exploratory Study on a ChatGPT-4-Based Tool for Cognitive Walkthrough Inspired Evaluations," in *International Conference on Advanced Visual Interfaces (AVI 2024)*, 2024, pp. 41:1–41:5. [Online]. Available: https://doi.org/10.1145/3656650.3656676

[48] M. Weninger, P. Grünbacher, E. Gander, and A. Schörgenhumer, "Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study," *Proc. ACM Hum. Comput. Interact.*, vol. 4, no. EICS, pp. 75:1–75:37, 2020. [Online]. Available: https://doi.org/10.1145/3394977

[49] J. Nielsen, *Usability Engineering*. Academic Press, 1993.

[50] T. R. G. Green, "Cognitive Dimensions of Notations," in *Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V*, 1989, pp. 443–460. [Online]. Available: http://dl.acm.org/citation.cfm?id=92968.93015

[51] A. F. Blackwell, C. Britton, A. L. Cox, T. R. G. Green, C. A. Gurr, G. F. Kadoda, M. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young, "Cognitive Dimensions of Notations: Design Tools for Cognitive Technology," in *International Conference on Cognitive Technology: Instruments of Mind (CT)*, 2001, pp. 325–341. [Online]. Available: https://doi.org/10.1007/3-540-44617-6_31

[52] A. Blackwell and T. Green, "Chapter 5 - notational systems—the cognitive dimensions of notations framework," in *HCI Models, Theories, and Frameworks*, ser. Interactive Technologies. Morgan Kaufmann, 2003, pp. 103 – 133. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9781558608085500058

[53] A. F. Blackwell, "Cognitive Dimensions of Notations," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2005, p. 3. [Online]. Available: https://doi.org/10.1109/VLHCC.2005.26

[54] J. Bennedsen and M. E. Caspersen, "Failure rates in introductory programming," *ACM SIGCSE Bull.*, vol. 39, no. 2, pp. 32–36, 2007. [Online]. Available: https://doi.org/10.1145/1272848.1272879

[55] C. Watson and F. W. B. Li, "Failure rates in introductory programming revisited," in *Innovation and Technology in Computer Science Education Conference (ITiCSE)*, 2014, pp. 39–44. [Online]. Available: https://doi.org/10.1145/2591708.2591749

[56] J. Bennedsen and M. E. Caspersen, "Failure rates in introductory programming: 12 years later," *Inroads*, vol. 10, no. 2, pp. 30–36, 2019. [Online]. Available: https://dl.acm.org/citation.cfm?id=3324888