Dynamic Visualization of Java Stream Operations for Education Using Marble Diagrams

Sebastian Stiegler
Institute for System Software
Johannes Kepler University
Linz, Austria

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

***Abstract*—Java Streams provide a declarative abstraction for data processing, but their lazy evaluation semantics make runtime behavior difficult to observe. Traditional debuggers are poorly suited for reasoning about element-level data flow, execution order, and transformation effects in Stream pipelines, particularly for novice programmers. We present a trace-driven, execution-centric visualization approach that makes Java Stream execution explicit using dynamic *Marble Diagrams*. We describe Marble Diagrams as a visual notation, introduce a reading strategy, and show how they can be reconstructed automatically from runtime traces. We realize this approach as an extension of the educational debugger *JavaWiz* and demonstrate its applicability through representative usage scenarios.**

***Index Terms*—Java Streams, Program Visualization, Software Education, Debugging, Marble Diagrams**

I. Introduction

Software development education presents unique challenges stemming from the abstract syntax and semantics of programming concepts [1], [2]. Students often struggle to understand the dynamic behavior of programs, making it difficult for them to clearly understand what the program does [3]–[5]. Previous research also shows that misunderstandings about program execution are widespread among novices [6]–[8]. This is especially problematic when program behavior is not directly visible in the source code, forcing learners to imagine the execution steps and data changes [9]. This could make it harder for novices to understand the program and reduce motivation, negatively impacting learning outcomes [10]. Visualizations have been shown to significantly improve learning outcomes in computer science [11]. By making abstract program behavior more concrete, they help novices build better mental models of program execution [12], improve their problem-solving skills, and increase their engagement [13], [14]. This supports cognitive load theory, which suggests that good visual designs can enable more efficient learning [15].

Nevertheless, it is difficult to predict the dynamic behavior of a program using only static code, especially in the case of declarative or library-based abstractions [9], [16]. A good example is the Java Stream API, which enables developers to express data processing pipelines declaratively through composable operations such as `filter`, `map`, or `flatMap` [17]. While this supports concise and clear code, the underlying lazy evaluation and execution within the library code hides how elements are processed at runtime. Traditional step-based debuggers are poorly suited for such declarative dataflow abstractions [18], [19]. Although they display the control flow at the level of individual statements, they do not provide insight into the data flow, the execution order, or the pipeline operations [18]. Therefore developers, especially novices, often struggle to understand the Java Stream API [20].

To address this gap, we present an execution-centric visualization approach based on dynamic *Marble Diagrams* [21]. Marble Diagrams provide a visual representation of data flow, making execution order, pipeline operations, and data origins visible [22], [23]. They are commonly built statically (and manually) in educational material, most prominently in RxMarbles [22]. While Marble Diagrams and trace-based program visualizations are established individually, we introduce an end-to-end technique that automatically reconstructs interactive Marble Diagrams from the concrete execution of user-written Java Stream pipelines. Our approach captures fine-grained execution data and applies a dynamic layout that makes execution order, element lineage, and transformation effects visible while preserving positional stability during incremental replay. Users can step forward and backward through execution to inspect processing order and perform root cause analysis. We implement both tracing and visualization as an extension of the educational debugger JavaWiz [24].

Our work encompasses the following main contributions:

- We describe Marble Diagrams as a visual language with defined key elements for reasoning about dataflow-oriented execution (Section III).
- We present a tracing technique to collect fine-grained execution traces for Java Stream operations that preserve operation order, semantics, and how elements are processed across pipeline operations (Section V).
- We present a trace-driven technique for dynamically reconstructing Marble Diagrams from Java Stream execution (Section VI).
- We demonstrate how this visualization supports educational and debugging tasks through representative usage scenarios (Section X).

II. Background and Motivation

The Java Stream API, introduced in Java 8 [17], [25], provides a declarative abstraction for processing sequences of elements through combinable operations. A Stream pipeline

This is the author's version of the work. The definitive version will be published in the Proceedings of the 14th IEEE Working Conference on Software Visualization (VISSOFT 2026).
A video accompanying this work can be found at https://ssw.jku.at/General/Staff/Weninger/Projects/JavaWiz/VISSOFT26/Video.mp4.
JavaWiz is open-source, its source code and binaries can be downloaded from https://javawiz.net.

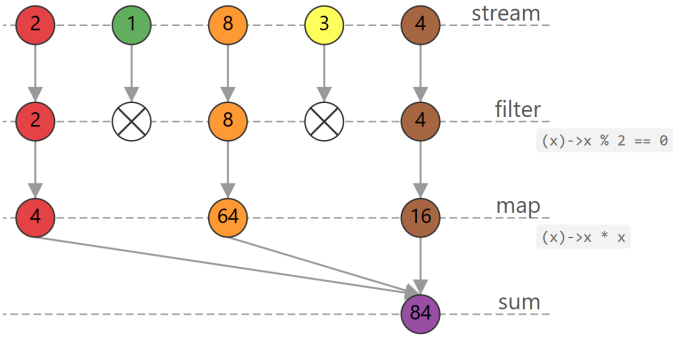

Fig. 1. A Marble Diagram illustrates execution order (left-to-right) and data lineage (top-to-bottom). Here, a Java Stream over numbers is shown, in which the pipeline operations `filter`, `map`, and `sum` are applied. 8 was the final number: it was not filtered out, squared, and then updated the sum.

consists of a source, intermediate operations, and a terminal operation [26]. Intermediate operations such as `filter`, `map`, and `flatMap` are *lazy*: they are composed into an execution plan that is only executed when a terminal operation such as `collect` or `count` is invoked. While this model enables concise pipeline definitions, it hides how elements are processed at runtime, decoupling the textual representation of a pipeline from its concrete execution behavior [19]. This mismatch introduces a cognitive challenge for developers, as the behavior of individual elements is not directly visible. Traditional debuggers show control flow at the level of individual statements [9] rather than at the stream-element level. Since Stream operations are lazily executed, stepwise debugging does not reveal how many elements are processed, which operations filter or transform elements, or how execution order emerges dynamically. This makes it difficult to understand the pipeline's behavior, especially for novice developers.

Empirical studies further highlight the prevalence of these challenges. Khatchadourian et al. [20] analyzed 34 Java projects and more than 700 code patches involving Stream pipelines, showing that developers predominantly use ordered pipelines with basic transformations, rarely exploit parallelization, and frequently introduce performance-related fixes. These findings suggest that Java Streams are often not used to their full potential. This could be due to a lack of tooling that helps people reason about dynamic Stream behavior. We assume that having better tool support could help educators to better teach Java Streams, which in turn could lead to better and more advanced Stream utilization. Complementing such usage studies, Rosales et al. [19] demonstrate the value of dynamic analysis for understanding runtime behavior by profiling Stream pipelines in terms of processor cycles and diagnosing performance bottlenecks in real-world workloads. Together, these results emphasize that making Stream execution behavior observable and visually recognizable is relevant not only for learning but also for practical performance engineering.

Taken together, the declarative structure of Stream pipelines, their lazy evaluation semantics, and empirical evidence from usage and profiling studies motivate the need for techniques that make Java Stream runtime behavior visible. By showing the execution order, intermediate results, and the relationships between elements, visualizations can help connect the code with what actually happens during execution.

III. The Marble Diagram Metaphor: A Primer

Marble Diagrams are a visual notation for visualizing data flow in reactive programming [22], [23], [27]. They represent a stream pipeline as a two-dimensional visual notation, where one dimension captures the sequence of pipeline operations and the other represents the execution order of data elements (see Figure 1). Its cognitive effectiveness is guided by principles such as semantic transparency, consistency, and perceptual discriminability [28]. They depict streams as sequences of discrete visual tokens (*marbles*) that traverse pipeline operations and are commonly used to explain stream operators in educational material and online documentation [22]. Despite their popularity, Marble Diagrams are typically presented as informal, static illustrations without explicit interpretation guidelines. Previous research in program visualization shows that the absence of a well-defined visual language can increase cognitive load and hinder systematic analysis [16], [29].

In this section, we present the various visual elements a Marble Diagram is made of, followed by a presentation of various techniques to read and reason about such a diagram.

A. Visual Notation

To ensure our design is built on a theoretical foundation, we follow the *Cognitive Dimensions of Notations* framework [30], [31], which provides guidelines for creating understandable visual representations and has been used in previous work to evaluate interactive program comprehension tools [32]. In our case, the notation focuses on data flow and execution order being immediately visible, making it easy to understand how elements move through the pipeline. To achieve this, we define the following visual components, also visible in Figure 1:

- **Lanes (Pipeline Operations):** Each horizontal lane represents a pipeline operation (e.g., `map`, `filter`, `flatMap`, or terminal operations) and marbles on these lanes show the stream state after that operation.
- **Marbles (Elements):** Marbles represent individual elements after each pipeline operation and can have value, identity, and/or type information.
- **Links (Lineage):** Directed vertical links connect marbles across lanes to represent lineage, including one-to-one, one-to-many, or many-to-one relationships. This supports lineage-oriented reasoning [33].
- **Horizontal Ordering (Execution Order):** Since spatial ordering supports comprehension of dynamic behavior [9], the horizontal axis encodes the relative order in which elements are processed rather than wall-clock time.

B. Reading and Analysis Strategies

Understanding how elements are transformed or filtered within a stream pipeline is difficult without an explicit representation of their relationships. Marble Diagrams make these

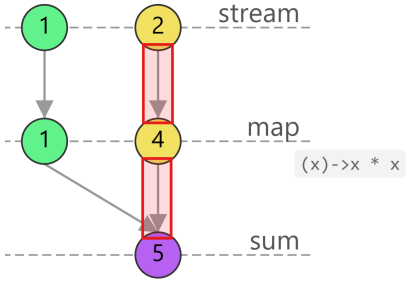

Fig. 2. Marble Diagram showing visible lineage paths across multiple pipeline operations. The diagram illustrates how individual elements can be traced through filtering, mapping, and aggregation without explicit interaction.

relationships explicitly visible by linking elements. To interpret these visual representations, users rely on complementary reading and analysis strategies [29]:

1) **Horizontal Scanning:** Reading a lane from left to right reveals how the number of elements changes at a specific stage. Missing elements or crossed-out marbles indicate that values have been removed by operations such as `filter` or `distinct`, while several consecutive elements may originate from a single input element, as in `flatMap`. This allows direct observation of how operations affect the size and structure of the pipeline.
2) **Vertical Slicing:** Following a marble across lanes reveals its lineage and transformation history from source to result, making it possible to trace how individual elements are processed throughout the pipeline.
3) **Backward Lineage Exploration:** Following links from a result back to its source helps with root cause analysis. For example, one may trace backwards to find out why a specific value appears at a specific stage or which input elements have contributed to a result. Since lineage links remain visible after they are revealed, relationships can be inspected while the visualization evolves step by step.
4) **Forward Lineage Exploration:** One can start from an input element and follow it step by step to see how the element is transformed by operations such as `map` or `flatMap`, and whether it is filtered out or disappears. This helps to understand how individual elements behave without losing sight of the overall process.

The first two strategies support stage-oriented reasoning, the latter two enable element-oriented reasoning. Together, they help users answer common questions about stream execution.

Building on these strategies, Marble Diagrams make value transformations directly visible. Differences between linked marbles show the effects of operations such as `map` or `flatMap`, including structural changes such as removing, extending, or reordering elements. This also makes visible how multiple elements contribute to an aggregated result. Figure 2 demonstrates how lineage links form connected paths through multiple pipeline operations, revealing the history of an element without the need for specific user interaction.

C. Role of the Metaphor in Dynamic Analysis

Based on execution data, Marble Diagrams make the runtime behavior of a Stream pipeline directly visible, which is difficult to observe with traditional debuggers alone. Compared to text-based debuggers, this provides a more direct look into the data flow [9], [16]. Software visualizations, including our Marble-Diagram-based Java Stream visualization, often strive to provide insights into fine-grained detail while at the same time providing an overview of the whole system [34]–[37].

IV. Approach Overview

This section provides a conceptual overview of our approach for dynamically visualizing Java Stream execution using Marble Diagrams. The system takes Java source code as input, traces the execution of Java Stream operations therein, and turns the result into an interactive visualization.

As shown in Figure 3, the system consists of three stages: *Static Analysis and Instrumentation*, *Execution and Trace Collection*, and *Visualization and Interaction*, a common design principle among trace-based analysis and visualization tools [21], [38]–[43].

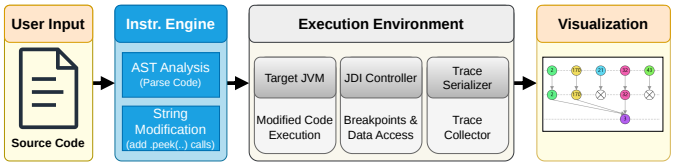

Fig. 3. System Overview: Source code is first instrumented, then executed and traced at run time, and finally transformed into interactive Marble Diagrams.

a) Static Analysis and Instrumentation: First, the system analyzes the source code to detect Stream pipelines by traversing the Abstract Syntax Tree (AST) and assigning each operation a unique identifier. Based on this information, tracing hooks are injected at well-defined points to observe execution without altering program semantics (see Section V). Restricting instrumentation to Stream-related constructs keeps runtime overhead manageable.

b) Execution and Trace Collection: After instrumentation, the program is executed normally on the JVM. Due to lazy evaluation, tracing events are emitted only when a terminal operation initiates execution. As elements flow through the pipeline, our tracing hooks record how elements propagate through the pipeline.

c) Visualization and Interaction: The visualization component transforms an execution trace into our visualization model (see Section VI), builds the individual visual representation for each entity (see Section VII) and finally applies a layout algorithm (see Section VIII) to arrange all elements. The resulting Marble Diagram can then be explored interactively by replaying the execution step by step (see Section IX).

V. Trace Collection via Stream Hooks

Dynamic visualization of Java Stream execution requires fine-grained runtime observations at the level of individual elements. To capture execution behavior without relying on

heavyweight debugging interfaces such as the Java Debug Interface (JDI) [44], [45], our approach instruments Stream pipelines directly at the source level using the Stream API's native `peek()` operation. This enables lightweight trace collection while preserving original pipeline semantics.

The `peek()` Instrumentation Strategy

Tracing logic is injected directly into Stream pipelines so that observations occur exactly when elements flow through operations. We leverage `peek()`, a debugging-oriented intermediate operation, to observe elements without modifying values or control flow. `peek()` calls are inserted before and after each operation, emitting structured trace events that record how elements enter and leave operations.

Conceptually, intermediate operations such as `filter(p)` are transformed as shown in Listing 1.

Listing 1. Capturing intermediate operation input and output.

```
.peek(x -> trace("IN", x, "filter", opId, sId, param))
.filter(person -> person.age >= 18)
.peek(x -> trace("OUT", x, "filter", opId, sId, param))
```

Stream creation (Listing 2) and terminal operations (Listing 3) are traced using `START` and `END` events.

Listing 2. Capturing stream creation operations.

```
persons
  .stream()
  .peek(x -> trace("START", x, "stream", opId, sId, ""))
```

Listing 3. Capturing terminal stream operations.

```
.peek(x -> trace("END", x, "count", opId, sId, ""))
.count()
```

Because `peek()` becomes part of the pipeline, tracing respects lazy evaluation semantics and emits events strictly in runtime execution order, ensuring that traces reflect actual behavior rather than inferred control flow. Each injected `peek()` call emits a structured runtime event forming the sole input for visualization. Each event contains:

- **Direction:** One of `START`, `IN`, `OUT`, or `END`, indicating the element's relationship to the operation.
  - **START** marks the beginning of a Stream pipeline.
  - **IN** and **OUT** delimit intermediate operations. For example, `filter` produces `IN` events for all elements but `OUT` events only for those satisfying the predicate, whereas `flatMap` may generate multiple `OUT` events.
  - **END** marks terminal completion, either through element consumption or aggregated results.
- **Element:** The observed stream element.
- **Operation Metadata:** The operation name and a unique identifier derived from static analysis.
- **Stream ID:** Unique identifier for each pipeline to distinguish events across multiple pipelines.
- **Parameters:** Optional textual representations of lambda expressions or method references.

To reduce the need for the user to switch focus between visualization and source code, operation-specific parameters are attached to trace events to make the visualization more self-contained. Extracted from the abstract syntax tree, these parameters are typically lambda expressions or method references. This enables us to annotate stages with the relevant code fragments (see Figure 4).

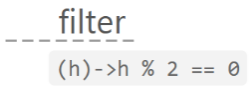

Fig. 4. Example of a lambda expression extracted from the abstract syntax tree to annotate a filter operation in the visualization.

VI. Visualization Data Model

While the runtime tracer described in Section V records execution events chronologically, effective visualization requires a structural representation that supports layout, interaction, and analysis. Prior work on trace-based software visualization emphasizes that raw execution traces must be transformed into higher-level graph representations to enable visual reasoning [21], [46].

Accordingly, the frontend does not operate directly on the raw event log. Instead, an intermediate processing layer transforms the trace into a graph-based data model composed of three primary entities: *Lanes*, *Marbles*, and *Links*, as summarized in Table I.

TABLE I
Data and Layout Model Structure for Marble Diagrams

Structure	Field	Type	Description
Lane	type	String	Name of the operation (e.g., `filter`, `map`)
	elementType	String	Type of the elements at this stage, e.g., `int` or `Person`
	param	String	Extracted lambda expression (e.g., `"x < 5"`)
	y	Integer	Vertical position of the lane
Marble	id	Integer	Unique identifier of the data element
	label	String	Value or summary displayed inside the marble
	type	String	Runtime type used to determine the visual representation
	color	String	Visual color encoding
	step	Integer	Execution step for animation sequencing
	x / y	Integer	Assigned layout coordinates
Link	source	Integer	Identifier of the source marble
	target	Integer	Identifier of the target marble
	step	Integer	Execution step for animation sequencing

A. Lanes

Each *lane* defines the vertical structure of the diagram, corresponding to a Stream operation, including source and terminal stages. They capture the static pipeline structure and serve as the backbone for arranging dynamic execution data.

Each lane also stores operation-specific metadata derived from static analysis. The `param` field contains the extracted

textual representation of the operation's lambda expression (e.g., `"x -> x % 2 == 0"`). This enables us to annotate lanes with relevant code fragments. Furthermore, each lane is assigned a fixed vertical position. The increase of `y` between different lanes depends on `elementType`, since marbles for different `elementType` (e.g., `int` or `Person`) require different heights (see Section VII).

B. Marbles

Marbles represent the elements flowing through the pipeline. To support the reading strategies introduced in Section III, each marble encodes various structural and semantic attributes:

- **Positioning:** Explicit `x` and `y` coordinates determine placement, where `y` associates a marble with a pipeline operation and `x` reflects relative execution order. Coordinate computation is described in Section VIII.
- **Representation:** Visual attributes convey semantic information beyond position. The `label` property either contains a primitive value or information about the respective heap object. The `type` field records runtime data types and determines the visual shape. `color` facilitates tracing one specific object across stages. For example, Figure 5 shows a `Person` object that is mapped to its first name. Both nodes, the source and mapped values, have the same color since they represent the same entity, even though the `map` operation transforms the `Person` object to a `String`.
- **Interactivity:** The `step` attribute indicates when a marble becomes visible during replay, enabling incremental animation and exploration.

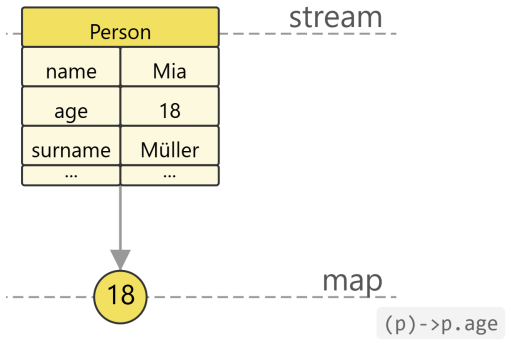

Fig. 5. Consistent colors convey the identity of objects across operations, even when the data type changes.

C. Links

Links represent the lineage between marbles across stages, making the data flow through the pipeline explicit. Links connect a source marble (or multiple source marbles) to a target marble (or multiple target marbles) and describe how elements are related across operations. The model supports three fundamental relationship types:

- **One-to-one:** A single input element produces a single output element (e.g., `map` – for example mapping a `Person` to its name (`String`)).
- **One-to-many:** A single input element expands into multiple output elements (e.g., `flatMap` – for example mapping a `Person` to its hobbies (`String[]`)).
- **Many-to-one:** Multiple input elements merge into a single output element (e.g., `count` – taking a stream of `String` objects and creating a single `int` count).

These links make it easy to follow how elements are connected across the pipeline and form the basis for lineage analysis in later sections.

VII. Type-Specific Visual Representations

While Marble Diagrams provide a consistent visual notation of data flows across Stream operations, a uniform representation (i.e., round marbles) of different value kinds (e.g., primitive types and objects) would lead to unclear differences between data types. Research in information visualization shows that visually distinct representations reduce cognitive load and support faster pattern recognition, especially for novice users [28], [29]. Building on this principle, our system employs *type-specific visual representations* that adapt the appearance of marbles to the data type they represent. Figure 6 summarizes these representations. The following subsections describe each strategy and its rationale.

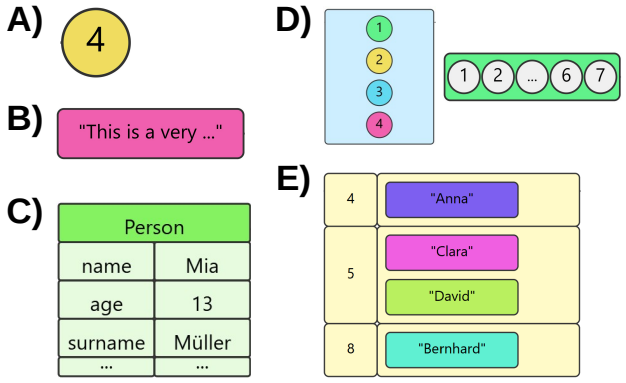

Fig. 6. Type-specific visual representations for different runtime data types. (A) Primitive values, (B) Strings, (C) Objects, (D) Lists, and (E) Maps. Each representation is designed to maximize perceptual discriminability and information density while preserving consistency with the Marble Diagram metaphor.

A. Primitive Values

Primitive data types such as integers, booleans, and characters use a classic round marble shape (Figure 6A). Since these values are simple, no additional structural information is required beyond the value itself. These simple labeled shapes support quick quantitative reasoning, comparison between operations, and enable efficient recognition tasks [29], [47].

B. Strings

Strings are visually distinguishable from primitives as rounded rectangles containing a textual preview (Figure 6B). Because strings often have semantic meaning, showing the value directly supports reasoning about filter and mapping

operations. To follow established visualization guidelines, long strings are truncated with an ellipsis to preserve readability when space is limited [29], [48].

C. Objects

Objects are visualized as structured containers, with the object type shown at the top followed by selected attributes and their respective values (Figure 6C). We prioritize attributes named `name` or `id`, as these attributes often act as meaningful identifiers, especially in teaching contexts. If more than three attributes exist, only the first three are shown, while the remaining ones are summarized using an ellipsis. This balances interpretability and visual clarity [28], [46].

D. Lists

Lists (Figure 6D) are represented differently depending on the context. *Terminal Lists*: When produced by terminal operations (such as `toList()`), lists are displayed as vertical containers showing all elements, supporting detailed inspection and reasoning about ordering and cardinality [16], [46]. *Intermediate Lists:* Between pipeline operations, the vertical space is limited. Lists created by intermediate operations therefore use a more compact representation showing up to five elements. Longer lists display the first two and last two entries with an ellipsis in between. This reduction preserves structural details while preventing visual overload [29], [48].

E. Maps

Maps are visualized as key–value tables (Figure 6E), where each row represents one mapping entry. Displaying keys and values side by side makes relationships explicit and supports reasoning about grouping and aggregation [46], [49].

F. Composability of Representations

All visualization types can be combined, allowing a consistent representation of nested data structures, such as object lists or maps containing lists. This design ensures clarity and visual coherence across different stream pipelines [28], [29].

VIII. Dynamic Marble Diagram Layout

The visualization data model introduced in Section VI provides a logical representation of stream execution using lanes, marbles, and links. To transform this model into a readable Marble Diagram, the system uses a dynamic layout algorithm that maps execution semantics to spatial structure.

This section describes how we aim for a layout that depicts the execution as accurately as possible, while still keeping it easy to understand.

A. Coordinate System

The layout is a two-dimensional coordinate system that reflects the visual notation introduced in Section III: Lanes, Marbles and Links. A marble's vertical position tells the user about the respective pipeline operation applied, while the horizontal axis shows the relative execution order.

Each pipeline operation is assigned a fixed vertical position, creating a horizontal lane that reflects the state of the stream after that operation. The order of the lanes follows the logical pipeline structure rather than the layout of the source code, i.e., multiple vertically separated lanes are created even if multiple stream operations appear on a single line of code. Horizontally, elements are arranged according to their order of appearance in the execution trace, which allows a comparison of the order and cardinality between operations.

B. Topological Layout Challenges

Stream operations often result in structural changes that cannot be represented by simple one-to-one mappings. The layout algorithm therefore handles four common situations that appear with expansion, reordering, filtering, and aggregation.

1) The Expansion Problem (Fan-Out): Operations such as `flatMap` generate multiple outputs from a single input. The layout assigns consecutive horizontal positions to the generated marbles and moves subsequent elements as needed, to ensure that the outgoing links clearly show the lineage (Figure 7). This preserves both ordering and lineage.

2) The Crossing Problem (Reordering): Reordering operations, such as `sorted`, may change the positions of elements. The layout retains the original positions in the input lane, while the reordered elements are placed at new horizontal positions. The cross-links explicitly show the permutation introduced (Figure 8), making changes in the order directly visible.

3) The Gap Problem (Filtering): Filter operations remove elements that do not satisfy a predicate. If the horizontal space for removed elements would be reduced, it would misrepresent the sequence semantics, so the layout preserves their positions and explicitly marks their absence using crossed placeholders (Figure 9). This maintains the alignment across lanes and allows immediate identification of removed elements.

4) The Reduction Problem (Aggregation): Terminal operations such as `count` or `collect` combine multiple inputs into a single result. The layout connects all participating marbles to a single output marble, which is placed in the terminal lane, while maintaining the lineage and indicating that the execution is complete (Figure 10). Aggregation marbles are the only ones that move over time: they update their position and label as new source marbles feed into them.

C. Incremental Layout and Animation

The layout is computed incrementally based on the `step` information from the data model, which tells the animation engine starting from which visualization step marbles and links should become visible. This enables a step-by-step animation that reflects the lazy execution of streams.

IX. Execution-Centric Stream Visualization

Stepping through a stream pipeline using a traditional debugger does not provide insight into the data flow at the element level. To address this, our system uses an *execution-centric visualization* [50], [51] approach: the Marble Diagram is generated after a Stream's terminal operation has completed, giving users a view of the complete history of what happened during execution.

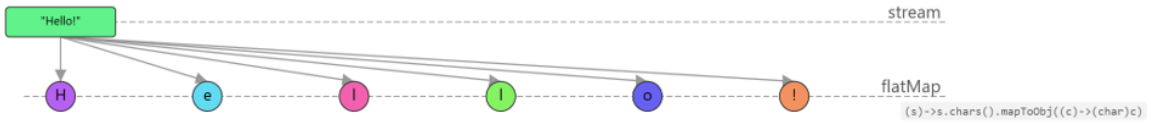

Fig. 7. Layout pattern for `flatMap`. A single marble expands into multiple output marbles arranged consecutively to preserve lineage and execution order.

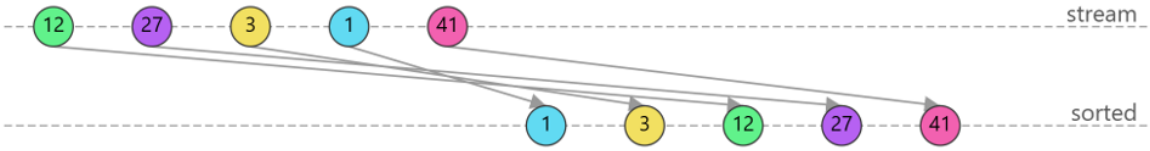

Fig. 8. Layout pattern for reordering operations. Crossing links encode changes in relative order between input and output lanes.

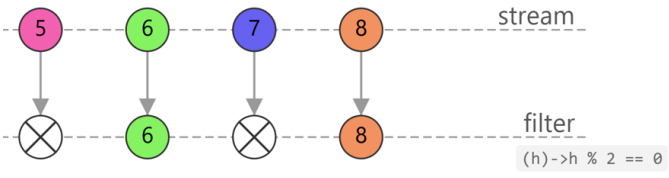

Fig. 9. Layout pattern for filtering operations. Removed elements preserve their horizontal position and are rendered as placeholders, maintaining alignment across lanes.

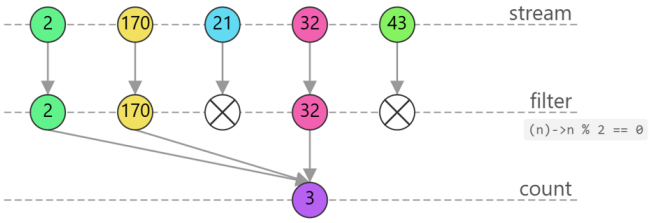

Fig. 10. Layout pattern for aggregation. Multiple input marbles converge into a single result marble, preserving lineage information.

A. Post-Execution Visualization

Once a Stream's execution finishes, the visualization is generated. At this point, the runtime trace contains a complete record of the processed elements, their execution order, and their lineage relationships. It has been shown that retrospective visualizations support understanding more effectively than strictly stepwise debugging [46].

B. Incremental Replay of Stream Execution

The visualization supports stepwise replay. Each step reveals one marble and its links, representing the observed execution order. Users can navigate forward or backward or jump to any step to see, step-by-step, how elements propagate through the pipeline. Figure 11 – Figure 13 illustrate sequential replay stages. Early steps show element introduction, while later steps reveal structural effects such as filtering and aggregation. Replay supports the reading strategies introduced in Section III: horizontal scanning reveals changes in cardinality, while vertical slicing allows tracing of individual elements.

C. Scope and Design Decisions

The visualization shows one stream pipeline at a time. Programs with multiple pipelines are visualized separately to avoid overload. Nested pipelines created by operations such as `flatMap` are not expanded (but planned as an "expert feature" as future work). Instead, the current visualization highlights the primary pipeline structure to ensure clarity, especially for novice users. Using more visual attributes can make the visualization richer, yet complex mappings should be used for complex tasks or expert systems only, since the mappings may become challenging to perceive [40], [52].

X. Educational Usage Scenarios

To illustrate the educational value of dynamic Marble Diagrams, we present three representative usage scenarios derived from common learning and debugging tasks of novice Java programmers. In addition to preliminary studies (as reported in Section XI), such illustrative evaluations are common in early-stage educational visualization research [9], [16].

A. Scenario 1: Understanding Nested Data Transformation

This scenario focuses on the analysis of nested data transformations that involve multiple intermediate operations. The Stream pipeline in Listing 4 generates a shopping list from a collection of families.

Listing 4. Novices often struggle to understand `flatMap`'s 1:n mapping.

```
1 List<Item> shoppingList = stream(families)
2   .flatMap(f -> stream(f.kids))
3   .flatMap(p -> stream(p.wishList))
4   .filter(i -> i.price < 200)
5   .sorted(comparing(i -> i.name))
6   .toList();
```

Using incremental replay, learners are able to observe how each family is expanded into its child elements `kids` and then into their wish-list items. One-to-many lineage links make the flattening process explicit, while the `filter` operation is visualized by marbles that terminate at the filter lane. The `sorted` operation introduces reordering, shown by crossing links. This allows learners to distinguish between structural transformation, selective removal, and reordering; concepts that are often difficult to derive from source code alone.

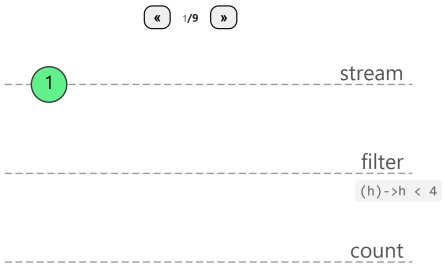

Fig. 11. Marble Diagram at step 1, showing initial element introduction.

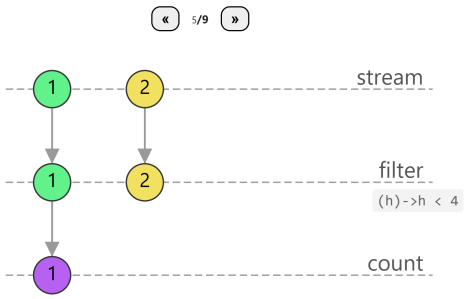

Fig. 12. Marble Diagram at step 5, illustrating intermediate operations and emerging structure.

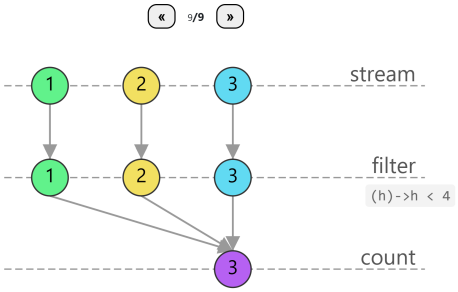

Fig. 13. Marble Diagram at step 9, showing the final state.

B. Scenario 2: Debugging Operation Ordering Errors

This scenario addresses a common mistake made by novices: placing expensive operations in suboptimal positions. For example, when `sorted` comes before `filter`, marbles that will be removed later are still part of the reordering. This leads to additional crossings and visual complexity. Reversing the order reduces the number of elements that reach the sorting operation, resulting in a simpler diagram.

This visual contrast allows novices to evaluate the impact on performance based on specific execution results rather than abstract explanations, reinforcing best practices in pipeline construction.

C. Scenario 3: Understanding Lazy Evaluation

Novices often have problems with the lazy execution model of Java Streams. In our system, the Marble Diagram only appears after a terminal operation has been called (e.g., line 3 in Listing 5), which visually illustrates that no execution takes place beforehand. The incremental replay then shows how elements flow through the pipeline, making it clear that elements only start to flow upon a terminal operation.

Listing 5. Java Streams are executed lazily.

```
1 Stream<Person> source = persons.stream();
2 IntStream ages = source.mapToInt(p -> p.age);
3 OptionalInt maxAge = ages.max(); /* The stream is only
4   executed when we reach this line (lazy evaluation) */
```

XI. Preliminary User Feedback

To gather initial feedback, we conducted a preliminary user evaluation as part of the undergraduate software engineering course “Software Development 2” at Johannes Kepler University Linz (JKU Linz), specifically a session in May 2026 on the Java Stream API. The session was split into two parts: a lecture (a 90-minute slide-based presentation of theoretical concepts) followed by an exercise lab (a 90-minute live-coding session). While the lecture remained unchanged, the tool was actively used during live-coding in the exercise lab (existing teaching material used in the exercise lab did not have to be adjusted for this). This allowed us to ask students to compare slide-based instruction with interactive tool usage (variable *SlideComparison*). We adopted the Technology Acceptance Model (TAM) [53], a well-established framework widely used to predict and explain the adoption of novel technologies based on two primary constructs: *Perceived Ease of Use (PEOU)* and *Perceived Usefulness (PU)*. Even though PU does not necessarily reflect real usefulness [54], it has been shown that PU has a strong influence on *usage intentions* of developer tools and methodologies [55], [56], i.e., *tool adoption*. Furthermore, recent studies demonstrate that TAM is particularly effective for assessing *student acceptance* of software visualizations in computer science education [57], [58]. The teaching material, the questionnaire, as well as anonymized student answers are made available as part of our replication package.

A. Demographics

63 students participated in the survey. The majority (76%) study computer science (others included AI, CS pedagogy, electronics, mechatronics, and medical engineering) and most (63.5%) were in their second semester. 84% participated for the first time in the course “Software Development 2”, 16% retook the course. 29% did not have any prior experience with JavaWiz, 33% were passive users in the past (i.e., watching lecturers using other JavaWiz visualizations), while 38% had actively used JavaWiz themselves before. JavaWiz is used in “Software Development 1”, the preceding course to “Software Development 2”. The 29% of students who did not have any experience with JavaWiz before most probably did not have to take “Software Development 1” due to prior experience (e.g., visiting a school with programming education).

B. Quantitative Results

Each participant was given five statements after the class in which the lecturers actively used the tool to explain Java Stream concepts. Students were informed that participation

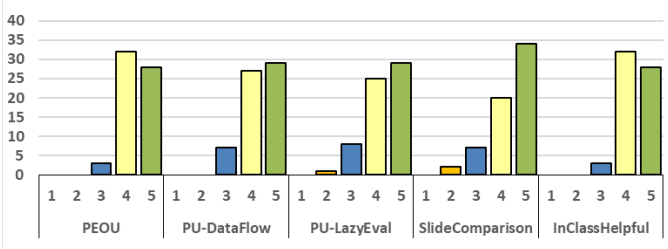

Fig. 14. Survey results based on the Technology Acceptance Model (1=Strongly Disagree, 5=Strongly Agree). Both Perceived Ease of Use (PEOU) and Perceived Usefulness (PU) received highly positive ratings.

was voluntary, responses were anonymous, and neither participation nor their answers affected grading. Students rated all statements on a standard 5-point Likert scale (1 = Strongly Disagree, 5 = Strongly Agree):

- *PEOU*: The visual notation (marbles, lanes, and links) was easy to understand.
- *PU-DataFlow*: The visualization helped me understand how elements are transformed and filtered in a Stream.
- *PU-LazyEval*: The step-by-step replay made it easier to understand the concept of lazy evaluation.
- *SlideComparison*: Compared to normal static presentation slides, the interactive visualization made Stream concepts easier to grasp.
- *InClassHelpful*: Overall, how helpful was the tool during the live exercise lab?

As laid out in Figure 14, the responses were highly positive. 47% of the respondents *strongly agreed* with statements, while a further 43% *agreed* with the statements. Only 1% *disagreed*, i.e., 3 responses out of 315.

C. Qualitative Results

We collected qualitative feedback via the open-ended question "*Do you have any suggestions to improve the tool or how the tool is used in lectures?*" at the end of the questionnaire.

Although the question aimed to identify improvement potential, various students used the open-ended question to report positive aspects of the tool. For example, one student replied "*Thank you so much for this tool. It definitely helped me so much to understand stream concepts much better then [sic] when someone only shows code and talks about it. The visual representation made it very clear for me how it works and what it does*," while another wrote "*The tool was not only very helpful but made a lasting impression for me regarding the topic streams. It presented the concept intuitively to the audience and gave a clear image how one can redraw some streams to find errors in the own logic.*" A few students reported possible improvements, including better filter highlighting for the `skip` and `distinct` operations (similar to the `filter` operation) as well as horizontally growing lanes (in the version used in the study lanes were always shown in their full width, even at the start of the animation and when not yet fully filled).

D. Threats to Validity and Future User Studies

This study was conducted as a preliminary evaluation in an instructor-led setting. Active, independent student-led debugging scenarios were out of scope. Although this setup introduces potential demonstration bias, it accurately reflects a primary use case of our approach: supporting educators in explaining complex declarative concepts in a classroom environment. To further mitigate this bias and limit the influence of any single instructor's teaching style and enthusiasm, the tool was demonstrated across multiple separate classes held by different lecturers not involved in this research. Furthermore, the risk of a novelty effect skewing the highly positive responses is considerably reduced, as 71% of the participating students were already familiar with the underlying JavaWiz debugger and trace-based debugging in general from a preceding course.

While our results provide strong initial evidence that novices *perceive* Marble Diagrams as easy to understand and useful, rigorous controlled experiments involving hands-on student tasks remain necessary future work to quantify the tool's *actual* impact on code comprehension and debugging efficiency, for example by comparing *task correctness*, *time-to-finish* and similar metrics across three groups: participants using no visualization, a standard debugger, or our visualization tool.

XII. Related Work

JavaWiz [24] and CrossCode [59] visualize concrete program executions beyond the information provided by conventional debuggers. JavaWiz combines visualizations of the heap, stack, control flow, and data structure evolution with time-travel navigation to support novice programmers [24]. CrossCode aggregates JavaScript execution according to syntactic structure and enables navigation across multiple levels of abstraction, using control-flow context, animations, trace paths, and color encodings to communicate program-state changes [59]. Neither system specializes in declarative data pipelines. Our work extends JavaWiz with a domain-specific visualization for Java Streams that exposes operation-level execution order, element transformations, and lineage through dynamically reconstructed Marble Diagrams.

Certain IDEs such as IntelliJ IDEA provide stream-specific debugging features, including the presentation of intermediate pipeline results [60], [61]. However, these intermediate results do not explicitly represent lineage or support systematic temporal exploration. In contrast, our approach reconstructs Marble Diagrams from runtime traces, enabling stepwise analysis of element flow independent of the development environment.

Static visual metaphors such as RxMarbles [22] are widely used in functional and reactive programming to illustrate operator semantics. While effective for explanation, these diagrams are manually constructed and not grounded in concrete execution, limiting their usefulness for debugging. Our approach applies this idea by deriving Marble Diagrams directly from runtime traces while preserving execution order and lineage.

More broadly, software visualization research has demonstrated that dynamic representations of execution can improve comprehension and debugging [24], [43], [51]. Trace-based

systems such as JIVE [62] or Jeliot [63] provide visual feedback on method calls, heap changes, and control flow beyond textual debuggers. Our work specializes this for dataflow semantics in pipeline-based code.

XIII. Limitations and Design Decisions

While the proposed system provides an interactive and educational view of Java Stream execution, certain limitations and design decisions reflect trade-offs and constrain its scope.

A. Handling of Infinite Streams

Our tool assumes finite Stream pipelines that terminate with bounded operations. Infinite streams created through constructs such as `Stream.generate` or `Stream.iterate` are only supported when explicitly truncated. Supporting unbounded execution would require abstraction strategies such as sampling or windowing to prevent uncontrolled visual growth, an open challenge for maintaining pedagogical clarity.

B. Performance and Trace Overhead

Although `peek()`-based instrumentation is lightweight, tracing large datasets introduces run-time overhead as well as increased memory utilization. While acceptable for educational scenarios, our main focus, this limits scalability for production workloads. This trade-off is common in educational trace-based visualization systems [21], [24].

C. Visual Scalability and Screen Space

Marble Diagrams expand horizontally with more elements and vertically with additional operations. For complex pipelines, this may exceed available screen space, requiring scrolling or zooming. Aggregation techniques would be necessary to support larger pipelines without reducing legibility.

XIV. Future Work

The proposed system opens several directions for future research and technical extension.

A. Performance-Oriented Visualization

In the future, the visualization could integrate timing and performance metrics. Recording operation-level processing times would enable performance analysis across three levels: per operation (e.g., a single `map` operation), at the stage level (e.g., all `map` operations across the whole stream), as well as at the element level (e.g., tracing how long a `filter-map-reduce` chain takes for an individual object). This may help students to improve their understanding of software performance [64], reducing the gap between educational visualization and performance debugging.

B. Alternative Visual Metaphors

While Marble Diagrams provide an intuitive representation of data flow, alternative metaphors such as Sankey diagrams [65], [66] could be explored. Systematic comparison across learning, debugging, and performance tasks may reveal their respective strengths and limitations.

C. Generalization to Other Dataflow Domains

The lineage-oriented nature of Marble Diagrams could be generalized beyond Java Streams to other dataflow domains, such as event-processing pipelines [67], log analytics [68], [69], or distributed architectures such as Kafka-based stream processing [70], [71]. Automatically generating trace-based Marble Diagrams across these domains represents a promising research direction.

D. Parallel Streams and Concurrency

Visualizing parallel Streams introduces challenges related to concurrency, non-deterministic ordering, and thread interleaving. Future work could investigate abstraction strategies to preserve clarity while representing parallel behavior.

E. Enhanced Interaction and Learning Support

Future extensions could provide richer interaction mechanisms, such as automated navigation to semantically significant execution steps or (AI-)guided walkthroughs [72] for common transformation patterns, further supporting self-directed learning and classroom use.

XV. Conclusion

Java Stream pipelines create an observability gap that limits traditional step-based debuggers, especially for novices.

This paper introduced a trace-driven, execution-centric visualization approach, implemented as an extension of the educational debugger JavaWiz, that makes Stream execution explicit through dynamic Marble Diagrams. We described Marble Diagrams as a visual notation with defined reading strategies that help users understand how elements flow through a pipeline. An interactive lineage analysis technique based on incremental replay supports both element-centric and operation-centric exploration without requiring direct source-code interaction. The resulting visualizations faithfully represent execution semantics, including filtering, expansion, reordering, and aggregation, and demonstrate how the approach supports understanding nested transformations, operation ordering, and lazy evaluation.

By bridging the gap between declarative Stream code and runtime behavior, dynamic Marble Diagrams provide a complementary perspective to existing debugging and visualization tools. While currently focused on educational contexts, the principles of execution-centric visualization and explicit lineage representation offer promising directions for future research on dataflow-oriented programming abstractions.

Data Availability Statement

To support open science and reproducibility, the source code of JavaWiz, the replication package for our study (the used teaching material, the questionnaire, and anonymized student answers), and the video are archived at https://doi.org/10.5281/zenodo.20730787.

References

[1] D. Fonte, D. d. Cruz, A. L. Gançarski, and P. R. Henriques, "A Flexible Dynamic System for Automatic Grading of Programming Exercises," in *2nd Symp. on Languages, Applications and Technologies (SLATE 2013)*, ser. Open Access Series in Informatics (OASIcs), vol. 29, Dagstuhl, Germany, 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. Robins, J. Rountree, and N. Rountree, "Learning and Teaching Programming: A Review and Discussion," *Computer Science Education*, 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.-M. Järvinen, "A study of the difficulties of novice programmers," in *10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2005)*, New York, NY, USA, 2005, p. 14–18. [Online]. Available: https://doi.org/10.1145/1067445.1067453

[5] J. Sorva, J. Lönnberg, and L. Malmi, "Students' Ways of Experiencing Visual Program Simulation," *Comput. Sci. Educ.*, vol. 23, no. 3, pp. 207–238, 2013. [Online]. Available: https://doi.org/10.1080/08993408.2013.807962

[6] T. Sirkiä and J. Sorva, "Exploring Programming Misconceptions: An Analysis of Student Mistakes in Visual Program Simulation Exercises," in *12th Koli Calling International Conference on Computing Education Research (Koli Calling '12)*. ACM, 2012, pp. 19–28. [Online]. Available: https://doi.org/10.1145/2401796.2401799

[7] V. Karavirta, A. Korhonen, and O. Seppälä, "Misconceptions in Visual Algorithm Simulation Revisited: On UI's Effect on Student Performance, Attitudes and Misconceptions," in *Conference on Learning and Teaching in Computing and Engineering (LaTiCE)*. IEEE Computer Society, 2013, pp. 62–69. [Online]. Available: https://doi.org/10.1109/LaTiCE.2013.35

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

[9] M. D. Storey, "Theories, tools and research methods in program comprehension: past, present and future," *Softw. Qual. J.*, vol. 14, no. 3, pp. 187–208, 2006. [Online]. Available: https://doi.org/10.1007/s11219-006-9216-4

[10] 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

[11] 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

[12] 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 *14th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2009)*, 2009, pp. 166–170. [Online]. Available: https://doi.org/10.1145/1562877.1562931

[13] 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

[14] 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

[15] J. Sweller, J. J. G. van Merrienboer, and F. G. W. C. Paas, "Cognitive Architecture and Instructional Design," *Educational Psychology Review*, vol. 10, no. 3, pp. 251–296, 1998. [Online]. Available: https://doi.org/10.1023/A:1022193728205

[16] C. Hundhausen, S. Douglas, and J. Stasko, "A Meta-Study of Algorithm Visualization Effectiveness," *Journal of Visual Languages and Computing*, vol. 13, no. 3, 2002.

[17] J. Nostas, J. P. S. Alcocer, D. E. Costa, and A. Bergel, "How Do Developers Use the Java Stream API?" in *21st International Conference on Computational Science and Its Applications (ICCSA 2021)*, ser. Lecture Notes in Computer Science, vol. 12955, 2021, pp. 323–335. [Online]. Available: https://doi.org/10.1007/978-3-030-87007-2_23

[18] B. Stein, L. Clapp, M. Sridharan, and B. E. Chang, "Safe stream-based programming with refinement types," in *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*, 2018, pp. 565–576. [Online]. Available: https://doi.org/10.1145/3238147.3238174

[19] E. Rosales, M. Basso, A. Rosà, and W. Binder, "Profiling and Optimizing Java Streams," *The Art, Science, and Engineering of Programming*, vol. 7, no. 3, pp. 1–39, 2023.

[20] R. Khatchadourian, Y. Tang, M. Bagherzadeh, and B. Ray, "An Empirical Study on the Use and Misuse of Java 8 Streams," in *Fundamental Approaches to Software Engineering (FASE 2020)*, ser. Lecture Notes in Computer Science, 2020, vol. 12076, pp. 97–118.

[21] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[22] "RxMarbles: Interactive diagrams for Reactive Extensions," https://rxmarbles.com, accessed 2026-01-09.

[23] E. Bainomugisha, A. L. Carreton, T. V. Cutsem, S. Mostinckx, and W. D. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 52:1–52:34, 2013. [Online]. Available: https://doi.org/10.1145/2501654.2501666

[24] M. Weninger, S. Grünbacher, and H. Prähofer, "JavaWiz: A Trace-Based Graphical Debugger for Software Development Education," in *33rd IEEE/ACM International Conference on Program Comprehension (ICPC 2025)*, 2025, pp. 147–158.

[25] N. Mehlhorn and S. Hanenberg, "Imperative versus Declarative Collection Processing: An RCT on the Understandability of Traditional Loops versus the Stream API in Java," in *44th IEEE/ACM International Conference on Software Engineering (ICSE 2022)*, 2022, pp. 1157–1168. [Online]. Available: https://doi.org/10.1145/3510003.3519016

[26] Oracle Corporation, "java.util.stream Package Summary," 2024, accessed: 2026-03-22. [Online]. Available: https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/stream/package-summary.html

[27] The Reactive Manifesto Authors, "The Reactive Manifesto," https://www.reactivemanifesto.org/, accessed 2026-01-09.

[28] D. L. Moody, "The Physics of Notations," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756–779, 2009.

[29] C. Ware, *Information Visualization: Perception for Design*, 3rd ed. Morgan Kaufmann, 2013.

[30] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *J. Vis. Lang. Comput.*, vol. 7, no. 2, pp. 131–174, 1996. [Online]. Available: https://doi.org/10.1006/jvlc.1996.0009

[31] A. F. Blackwell and T. R. G. Green, "A Cognitive Dimensions questionnaire optimised for users," in *12th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2000)*, 2000, p. 10. [Online]. Available: https://ppig.org/papers/2000-ppig-12th-blackwell/

[32] 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

[33] J. Cheney, L. Chiticariu, and W.-C. Tan, "Provenance in Databases: Why, How, and Where," *Foundations and Trends in Databases*, 2009.

[34] C. Armenti and M. Lanza, "Using Interactive Animations to Analyze Fine-grained Software Evolution," in *IEEE Working Conference on Software Visualization (VISSOFT 2024)*, 2024, pp. 36–47. [Online]. Available: https://doi.org/10.1109/VISSOFT64034.2024.00014

[35] M. André, M. Raglianti, A. Cleve, and M. Lanza, "Visualizing and Exploring Data Access in Microservices Using Interactive Treemaps," in *IEEE Working Conference on Software Visualization (VISSOFT 2025)*, 2025, pp. 36–46. [Online]. Available: https://doi.org/10.1109/VISSOFT67405.2025.00012

[36] M. Giannaccari, M. Raglianti, and M. Lanza, "Skylines: Visualizing Object-Oriented Software Systems Through Class Contours," in *IEEE Working Conference on Software Visualization (VISSOFT 2025)*, 2025,

pp. 64–68. [Online]. Available: https://doi.org/10.1109/VISSOFT67405.2025.00015

[37] M. Hansen, J. Bamberg, N. Baumann, and W. Hasselbring, "Semantic Zoom and Mini-Maps for Software Cities," in *IEEE Working Conference on Software Visualization (VISSOFT 2025)*, 2025, pp. 47–57. [Online]. Available: https://doi.org/10.1109/VISSOFT67405.2025.00013

[38] M. Weninger, L. Makor, E. Gander, and H. Mössenböck, "AntTracks TrendViz: Configurable Heap Memory Visualization Over Time," in *ACM/SPEC International Conference on Performance Engineering Companion (ICPE)*, 2019, pp. 29–32. [Online]. Available: https://doi.org/10.1145/3302541.3313100

[39] M. Weninger, L. Makor, and H. Mössenböck, "Memory Leak Visualization using Evolving Software Cities," *Softwaretechnik-Trends*, vol. 39, no. 4, pp. 44–46, 2019. [Online]. Available: https://fb-swt.gi.de/fileadmin/FB/SWT/Softwaretechnik-Trends/Verzeichnis/Band_39_Heft_4/SSP2019_Weninger.pdf

[40] ——, "Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor," in *IEEE Working Conference on Software Visualization (VISSOFT 2020)*, 2020, pp. 110–121. [Online]. Available: https://doi.org/10.1109/VISSOFT51673.2020.00017

[41] ——, "Heap Evolution Analysis Using Tree Visualizations," *Softwaretechnik-Trends*, vol. 40, no. 3, pp. 64–66, 2020. [Online]. Available: https://fb-swt.gi.de/fileadmin/FB/SWT/Softwaretechnik-Trends/Verzeichnis/Band_40_Heft_3/SSP2020_Weninger_Makor.pdf

[42] ——, "Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations," in *Conference on Smart Tools and Applications in Graphics (STAG 2020)*, 2020, pp. 63–75. [Online]. Available: https://doi.org/10.2312/stag.20201241

[43] T. Herber and M. Weninger, "Trace-Based Bytecode Interpreter Visualization for Compiler Construction Education," in *IEEE Working Conference on Software Visualization (VISSOFT 2025)*, 2025, pp. 13–24. [Online]. Available: https://doi.org/10.1109/VISSOFT67405.2025.00010

[44] D. J. Murray and D. E. Parson, "Automated Debugging in Java Using OCL and JDI," in *4th International Workshop on Automated Debugging (AADEBUG 2000)*, 2000. [Online]. Available: https://arxiv.org/abs/cs/0101002

[45] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)," in *International Seminar on Software Visualization (SoftVis 2001)*, ser. Lecture Notes in Computer Science, 2001, pp. 176–190. [Online]. Available: https://doi.org/10.1007/3-540-45875-1_14

[46] S. Diehl, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*, ser. Springer Series on Software Engineering. Springer, 2007.

[47] W. S. Cleveland and R. McGill, "Graphical Perception: Theory, Experimentation, and Application," *Journal of the American Statistical Association*, vol. 79, no. 387, pp. 531–554, 1984.

[48] J. Bertin, *Semiology of Graphics*. University of Wisconsin Press, 1983.

[49] J. H. Larkin and H. A. Simon, "Why a Diagram is (Sometimes) Worth Ten Thousand Words," *Cog. Sci.*, vol. 11, no. 1, pp. 65–100, 1987.

[50] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk, "Execution trace analysis through massive sequence and circular bundle views," *J. Syst. Softw.*, vol. 81, no. 12, pp. 2252–2268, 2008. [Online]. Available: https://doi.org/10.1016/j.jss.2008.02.068

[51] A. Ernstsson, E. Frankell, and C. W. Kessler, "Interactive Performance Visualization and Analysis of Execution Traces for Pattern-Based Parallel Programming," *Int. J. Parallel Program.*, vol. 53, no. 5, p. 29, 2025. [Online]. Available: https://doi.org/10.1007/s10766-025-00805-3

[52] D. Limberger, W. Scheibel, J. Döllner, and M. Trapp, "Advanced Visual Metaphors and Techniques for Software Maps," in *12th International Symp. on Visual Information Communication and Interaction (VINCI 2019)*, 2019, pp. 11:1–11:8. [Online]. Available: http://doi.org/10.1145/3356422.3356444

[53] F. D. Davis, "Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology," *MIS Q.*, vol. 13, no. 3, 1989. [Online]. Available: http://misq.org/perceived-usefulness-perceived-ease-of-use-and-user-acceptance-of-information-technology.html

[54] D. A. Norman, "Some Observations on Mental Models," in *Mental Models*. Psychology Press, 1983.

[55] C. K. Riemenschneider, B. C. Hardgrave, and F. D. Davis, "Explaining Software Developer Acceptance of Methodologies: A Comparison of Five Theoretical Models," *IEEE Trans. Software Eng.*, vol. 28, no. 12, pp. 1135–1145, 2002. [Online]. Available: https://doi.org/10.1109/TSE.2002.1158287

[56] D. Russo, "Navigating the Complexity of Generative AI Adoption in Software Engineering," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, pp. 135:1–135:50, 2024. [Online]. Available: https://doi.org/10.1145/3652154

[57] E. Kühlmann, S. Hamer, and C. Quesada-López, "Software Visualization using the City Metaphor: Students' Perceptions and Experiences," in *49th Latin American Computer Conference (CLEI 2023)*, 2023, pp. 1–10. [Online]. Available: https://doi.org/10.1109/CLEI60451.2023.10346099

[58] S. I. Malik, R. Mathew, R. M. Tawafak, G. A. Farsi, and A. Al-Sideiri, "Investigating the impact of the OOP-SOLVE application on the user's behavior using the technology acceptance model in the programming course," *Frontiers Comput. Sci.*, vol. 7, 2025. [Online]. Available: https://doi.org/10.3389/fcomp.2025.1510577

[59] D. Hayatpur, D. Wigdor, and H. Xia, "CrossCode: Multi-level Visualization of Program Execution," in *Conference on Human Factors in Computing Systems (CHI)*. ACM, 2023, pp. 593:1–593:13. [Online]. Available: https://doi.org/10.1145/3544548.3581390

[60] JetBrains, "Debugging Streams – JetBrains Guide," https://www.jetbrains.com/guide/java/tips/debugging-streams/, accessed: 2026-01-02.

[61] JetBrains Documentation, "Analyze Java Stream Operations," https://www.jetbrains.com/help/idea/analyze-java-stream-operations.html, accessed: 2026-01-02.

[62] P. V. Gestwicki and B. Jayaraman, "Methodology and Architecture of JIVE," in *ACM Symp. on Software Visualization (SOFTVIS)*, 2005, pp. 95–104. [Online]. Available: https://doi.org/10.1145/1056018.1056032

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

[64] D. Krishnamurthy, "Teaching Software Performance Evaluation to Undergrads: Lessons Learned and Challenges," *SIGMETRICS Perform. Evaluation Rev.*, vol. 51, no. 2, pp. 50–52, 2023. [Online]. Available: https://doi.org/10.1145/3626570.3626589

[65] H. Miyachi and M. Kimura, "Study on the Visualization of Recycling Processes Using Sankey Diagrams and Their Effects," in *27th International Conference on Network-Based Information Systems (NBiS 2024)*, 2024, pp. 347–355. [Online]. Available: https://doi.org/10.1007/978-3-031-72325-4_34

[66] Z. Xu, B. Zhou, Z. Yang, X. Yuan, Y. Zhang, and Q. Lu, "NeatSankey: Sankey diagrams with improved readability based on node positioning and edge bundling," *Comput. Graph.*, vol. 113, pp. 10–20, 2023. [Online]. Available: https://doi.org/10.1016/j.cag.2023.05.001

[67] A. Bédard and S. Hallé, "Formal verification for event stream processing: Model checking of BeepBeep stream processing pipelines," *Inf. Comput.*, vol. 293, p. 105058, 2023. [Online]. Available: https://doi.org/10.1016/j.ic.2023.105058

[68] D. Borysenkov, A. Vogel, S. Henning, and E. Pérez-Wohlfeil, "Analyzing Logs of Large-Scale Software Systems Using Time Curves Visualization," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2025)*, 2025, pp. 327–337. [Online]. Available: https://doi.org/10.1109/SANER64311.2025.00038

[69] L. Leko, S. Henning, A. Vogel, O. Ertl, and R. Rabiser, "Benchmarking Pattern Matching Strategies for Scalable Log Analytics," *Softwaretechnik-Trends*, vol. 45, no. 4, pp. 50–52, 2025. [Online]. Available: https://fb-swt.gi.de/fileadmin/FB/SWT/Softwaretechnik-Trends/Verzeichnis/Band_45_Heft_4/SSP25_paper_14.pdf

[70] G. Wang, L. Chen, A. Dikshit, J. Gustafson, B. Chen, M. J. Sax, J. Roesler, S. Blee-Goldman, B. Cadonna, A. Mehta, V. Madan, and J. Rao, "Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka," in *ACM SIGMOD International Conference on Management of Data (SIGMOD 2021)*, 2021, pp. 2602–2613. [Online]. Available: https://doi.org/10.1145/3448016.3457556

[71] D. Chen, S. Henning, K. Matteussi, and R. Rabiser, "Performance Optimization in Stream Processing Systems: Experiment-Driven Configuration Tuning for Kafka Streams," in *17th ACM/SPEC International Conference on Performance Engineering Companion (ICPE Companion 2026)*, 2026, pp. 154–159. [Online]. Available: https://doi.org/10.1145/3777911.3800636

[72] M. Weninger, "Towards Guided Omniscient Debugging in Education using Pedagogical Execution Traces (PETs)," in *4th Workshop on Future Debugging Techniques (DEBT 2026)*, 2026. [Online]. Available: https://ssw.jku.at/General/Staff/Weninger/Papers/Weninger_DEBT_26_Preprint.pdf