JOHANNES KEPLER
UNIVERSITY LINZ

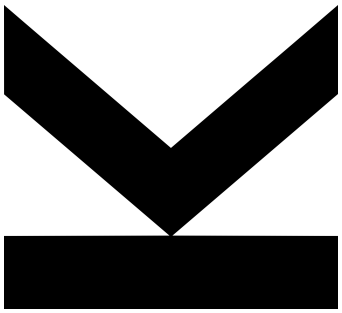
Author
Alexander Do, BSc
k11910655

Submission
Institute for
System Software

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

September 2025

Capture & Replay of JavaWiz Applications in GraalVM Espresso



Master Thesis

to obtain the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

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

Kurzfassung

Debugging- und Visualisierungstools spielen eine wichtige Rolle in der Softwareentwicklungsausbildung in der Informatik, da sie unerfahrenen Programmierern helfen, das Verhalten von Programmen zu verstehen. In diesem Zusammenhang hat sich JavaWiz, ein visueller Debugger für Java, als effektiv erwiesen. Jedoch fehlte bisher die Unterstützung für die Aufzeichnung und Wiedergabe von Ausführungen, was die Reproduzierbarkeit und die Zusammenarbeit einschränkt.

Diese Arbeit stellt eine Erweiterung von JavaWiz vor, die einen Capture-and-Replay-Mechanismus umfasst, der auf dem GraalVM Espresso-Interpreter und dem Truffle-Framework basiert, welche beide von Oracle Labs entwickelt wurden. Das vorgeschlagene System zeichnet nichtdeterministische Ereignisse wie Benutzereingaben, System-Calls oder zeitabhängige Operationen während der Ausführung auf. Anschließend werden diese Ereignisse deterministisch wiedergegeben, um die Reproduktion identischen Verhaltens sicherzustellen. Das System ermöglicht auch die Verzweigung von Programmausführungen, um alternative Ausführungspfade zu analysieren und zu testen. Darüber hinaus untersucht die Arbeit die Machbarkeit von Zustands-Snapshots, um Programme an beliebigen Punkten zu laden und fortzusetzen. Ein weiterer Punkt dieser Arbeit ist die Kompilierung des Systems zu JavaScript mit GraalVM Web Image, wodurch JavaWiz vollständig browserbasiert ausgeführt werden kann. Dies erleichtert den Zugriff auf JavaWiz und unterstützt die interaktive Nutzung in Unterrichtsumgebungen, ohne dass eine Installation erforderlich ist. Die Implementierungsergebnisse zeigen die pädagogischen Vorteile dieses Ansatzes, darunter Reproduzierbarkeit, die Möglichkeit, Debugging-Sitzungen zu teilen, und die Möglichkeit, von Vortragenden vorbereitete Beispiele im Unterricht wiederzugeben.

Abstract

Debugging and visualization tools play a vital role in software development education in computer science by helping novice programmers understand program behavior. In this context, JavaWiz, a visual debugger for Java, has proven effective. But JavaWiz lacked support for recording and replaying executions, limiting reproducibility and collaboration.

This thesis presents an extension to JavaWiz, incorporating a Capture and Replay mechanism built on the GraalVM Espresso interpreter and the Truffle framework, both developed by Oracle Labs. The proposed system records nondeterministic events, such as user input, system calls, or time-dependent operations during execution. Subsequently, these events are replayed in a deterministic manner to ensure the reproduction of identical behaviour. The system also enables branching of program executions to analyze and test alternative execution paths. Furthermore, the work explores the feasibility of state snapshots to load and resume programs at arbitrary points. Another aspect of this work is compiling the system to JavaScript using GraalVM Web Image, which allows JavaWiz to be executed entirely browser-based. This facilitates access to JavaWiz and supports interactive use in teaching environments without requiring installation. The implementation results demonstrate the educational benefits of this approach, including reproducibility, the ability to share debugging sessions, and the opportunity to replay lecturer-prepared examples in classrooms. This contribution advances JavaWiz into a more comprehensive and flexible teaching and debugging tool, supporting both individual learning and collaborative analysis of Java programs.

Contents

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Goals . . . 2
1.3 Structure of the Thesis . . . 3

2 Background **5**
2.1 JavaWiz . . . 5
2.2 GraalVM . . . 9
2.2.1 Truffle . . . 10
2.2.2 Espresso . . . 11

3 Capture and Replay Approach **13**
3.1 Overview . . . 13
3.2 Design Considerations for Java Replayability . . . 15
3.3 Role of Truffle and GraalVM Espresso . . . 16
3.4 Event Recording and Replay . . . 16
3.4.1 Why Recording is Necessary . . . 16
3.4.2 Handling Nondeterministic Operations . . . 17

4 User Interface and Use Cases **20**
4.1 User Interaction . . . 20
4.2 Visual Design and Interaction Features . . . 26
4.2.1 Program execution trace tree . . . 26
4.2.2 Sharing . . . 28
4.3 Examples of Educational Use Cases . . . 29
4.3.1 Scenario 1: Teaching Exception Handling . . . 30
4.3.2 Scenario 2: Exploring Input-Driven Behavior through Replay Branching . . . 32
4.3.3 Scenario 3: Trace Sharing and Remote Instruction . . . 34

5 Architecture **36**
5.1 Overview . . . 36
5.2 Trace Log . . . 37
5.3 Virtual File Environment . . . 38

5.4 Tracer 38

6 Implementation **40**
6.1 Core Components 40
6.1.1 Trace Log Implementation 42
6.1.2 Virtual File Environment Implementation 44
6.1.3 Tracer Implementation 50
6.2 Intercepting with `@Substitution` 56
6.3 Step Back to Previous States 60
6.4 Sharing and Collaboration 66

7 Discussion **68**
7.1 Limitations 68
7.1.1 Multithreading and Concurrency 68
7.1.2 Networking I/O 70
7.2 Espresso Continuations 73

8 Related Work **78**
8.1 DRD of SoftPLC Programs 78
8.2 Omniscient Debugging 79
8.3 BugVis 79
8.4 Undo 80
8.5 IntelliTrace 80

9 Conclusion and Future Work **82**

List of Figures

2.1 Overview of the JavaWiz backend processing steps. Black arrows indicate processing steps during compilation and initialization. Red arrows indicate processing steps during execution. 7
2.2 Screenshot of JavaWiz execution. 9
2.3 Overview of the GraalVM architecture [12]. 10
2.4 Java on Truffle in the GraalVM architecture [15]. 12

3.1 Deterministic Replay Debugging. The left side shows the recording phase (System 1), where the program is executed in JavaWiz and nondeterministic events are captured. The right side shows the replay phase (System 2), where the recorded data is used to reproduce the original execution behavior. 14
3.2 Execution of a simple input/output program in JavaWiz paused at line 3. At this point, the heap contains only the empty `String[] args`, and recorded user input (`"Lena"`) will be replayed later without re-prompting. 19

4.1 The user can start the recording from the toolbar. 21
4.2 Toggled Execution trace tree showing the initial program state. 22
4.3 Execution trace tree evolves. 23
4.4 Branching from a trace state. 24
4.5 Snapshots can be saved using share option from the toolbar. 24
4.6 The user can start the replay from the toolbar. 25
4.7 Loaded replay session with execution trace tree. 26
4.8 Interactive program execution trace tree in JavaWiz. 28
4.9 Popup modal dialog for sharing in JavaWiz. 29
4.10 Screenshot of JavaWiz Replay Scenario 1. 31
4.11 Screenshot of JavaWiz Replay Scenario 2. 33

5.1 Architecture diagram showing interaction of components. 37

6.1 Class diagram of the core components of the JavaWiz Trace System. 41
6.2 Replay with separate buffers: values are consumed from the replay buffer while the record buffer remains intact as the canonical source. 53
6.3 Tracing flow: replayed values are returned if available, otherwise the real operation is executed and optionally traced. 59

6.4 Workflow of the `step back` functionality in JavaWiz. The user selects an earlier state, the system deterministically replays execution up to that point, and execution continues from there. 61
6.5 Sequence diagram of the step-back execution. 62
6.6 Sharing workflow. User 1 records an execution and uploads the recording to the web server. User 2 fetches the recording and replays the execution. 67

7.1 Theoretical workflow of the `step back` functionality in JavaWiz realized with the Continuation API. The user selects an earlier state, the system loads the previous snapshot from this state and executes up to the selected state, and execution continues from there. 76

Listings

4.1 Example of user input parsing with `Integer.decode()` 30
4.2 Example of buggy permutations (causes stack overflow) 34

6.1 TraceEntry data structure. 42
6.2 Branch aware record and consume of values. 43
6.3 Persist and reload of the Buffer. 43
6.4 Registering the `memory://` scheme. 44
6.5 Routing open options to the store. 45
6.6 File store backed by a concurrent map. 46
6.7 Persist and reload the file store. 46
6.8 Grow-on-write semantics. 47
6.9 Truncating a file. 48
6.10 API for managing the underlying file store. 48
6.11 Scheme-aware path URIs. 49
6.12 Tracer execution mode handling 50
6.13 Tracer recording values when in record mode and reproducing them when replaying. 51
6.14 Copying the record buffer into a dedicated replay buffer. 53
6.15 Selective tracing of debuggee code only. 55
6.16 Declaration of System.currentTimeMillis() in the standard JDK. 56
6.17 Example: Substituting System.currentTimeMillis(). 56
6.18 Guest code that executes the substituted native method. 57
6.19 Tracer-aware substitution of System.currentTimeMillis(). 58
6.20 Handling a step-back request by setting stepBackInformation (undo trace) and restarting. 63
6.21 Advancing step-back metadata during replay. 63
6.22 StepBackTask: target condition for completing a step-back 64
6.23 VM-directed navigation to earlier states. 64
6.24 Dummy trace states are created. 65
6.25 Condition and completion for step back. 65

7.1 HTTP request failing when the server is no longer available 70
7.2 Sequential SQL statements in the main execution 71
7.3 Branch replay after stepping back 72

7.4 Capturing an authentication token from an HTTP response 73
7.5 Snapshotting with continuations (illustrative). 74

1 Introduction

In the context of software development education, visualization of program behavior and debugging plays a critical role in understanding program behavior and identifying bugs. While experienced developers may rely on traditional debuggers and logs to trace execution, novice programmers often struggle with abstract representations of runtime behavior of programs. To bridge this gap, tools have emerged that visualize runtime information in a more intuitive and accessible manner.

JavaWiz is a visual Java debugger developed at the Institute for System Software of the Johannes Kepler University Linz [1, 2]. It serves as a visualization tool specifically designed for educational purposes, aiming to simplify the debugging process in Java for students and novice programmers. JavaWiz provides multiple views designed to highlight different aspects of the execution of the program, such as a desk test that shows how variable values change over time, a graphical representation of the call stack and heap, and a flowchart that outlines the control flow. Additionally, JavaWiz offers specialized visual renderings for data structures such as arrays, linked lists, and trees, as well as a visualization for the input buffer.

The JavaWiz frontend is developed in Vue.js [3] with TypeScript and utilizes the D3.js [4] library for data driven visualizations. On the backend, JavaWiz relies on a Java-based execution engine to run programs in debug mode. Currently, there are two versions of the backend: one based on the Java Debug Interface (JDI) and another based on GraalVM Espresso, a Java bytecode interpreter implemented entirely in Java [5].

This thesis explores the development of an extended backend for JavaWiz, leveraging GraalVM Espresso to implement functionality for capture and replay (also referred to as recording and replaying) of Java applications. The core idea involves intercepting native method calls to capture input values, thereby enabling full replayability of application executions. In addition, the thesis investigates the possibility of taking runtime snapshots

of application states, which can later serve as the starting point for resumed or alternative executions.

1.1 Motivation

Debugging is not only a technical task but also a crucial pedagogical tool. For programming novices, understanding the cause and effect of code execution is essential for learning. However, conventional debugging tools often require a deep understanding of low-level runtime mechanics, which can overwhelm inexperienced users.

JavaWiz addresses this issue by providing an accessible and visual representation of the behavior of Java programs. Nevertheless, the current implementations of the backend have limitations. For example, while the JDI-based backend is capable to trace information on the execution state, it does not provide an Application Programming Interface (API) that supports recording or replaying executions. Similarly, GraalVM Espresso offers promising capabilities due to its interpretive nature and modular design, but lacks mechanisms for persistent state tracking and replay in the current JavaWiz integration.

Introducing a capture and replay feature into JavaWiz based on GraalVM Espresso offers significant advantages. It would allow users to reproduce specific execution scenarios, share executions with collaborators and resume analysis from any captured point. This not only supports the learning process, but also enhances the reproducibility and portability of debugging sessions in both educational and collaborative contexts.

By extending the Espresso backend to include these capabilities, this thesis contributes toward making JavaWiz a more powerful and flexible tool for teaching, debugging and collaborating in Java programming.

1.2 Goals

The primary goal of this thesis is to design and implement a capture and replay functionality for JavaWiz using the GraalVM Espresso backend. This enhancement aims to enable deterministic execution replay of Java applications, thereby expanding the capabilities of JavaWiz as a teaching and debugging tool.

To achieve this, the following sub-goals are defined:

- Recording mechanism: Develop a mechanism to intercept and record the execution of native method calls during program runtime. The goal is to capture relevant input data and nondeterministic interactions such as system calls, user input, and time dependent operations.
- Replay engine: Implement a replay system that accurately reproduces the recorded application behavior. This requires replacing native calls with their recorded counterparts and ensuring consistency across different execution environments.
- Snapshot functionality: Investigate the feasibility of capturing snapshots of the runtime state of the application, including call stack, heap and execution context as well as restoring the program from these snapshots at a later point.
- Showcase use cases: Showcase the developed system and its usability. Additionally, demonstrate its utility through practical scenarios, such as sharing recorded sessions or resuming classroom exercises from a known state.
- Web version: Using GraalVM Web Image to provide JavaWiz as a web application and in this way allow local execution in the web browser.

The overarching aim of this work is to contribute to the development of capable and user friendly educational tools that enhance the understanding of program behavior, especially for students learning object-oriented programming in Java.

1.3 Structure of the Thesis

This thesis is structured as follows:

- Chapter 2 provides the background information on JavaWiz, the GraalVM, the Truffle framework and Espresso, which are essential for understanding the implementation of the capture and replay functionality.
- Chapter 3 outlines the foundations of the capture and replay approach and the conception of the recording and replay workflow.

- Chapter 4 demonstrates the interface design and usage scenarios for the capture and replay functionality, detailing how users can interact with the system to record and replay executions.
- Chapter 5 describes the architecture of the system, detailing how it integrates with JavaWiz and GraalVM Espresso to achieve deterministic replay.
- Chapter 6 presents the implementation details, including the core components of the system, the mechanisms for capturing and storing execution data, and the replay engine.
- Chapter 7 discusses the limitations of the current implementation and potential future work to further enhance the capabilities of JavaWiz. In particular, discuss how snapshots could be realized based on the GraalVM Espresso Continuation API.
- Chapter 8 reviews related work in the field of debugging and program visualization, highlighting how the capture and replay functionality in JavaWiz builds upon and extends existing approaches.
- Finally, Chapter 9 concludes the thesis by summarizing the contributions made and reflecting on the impact of this work on programming education.

2 Background

This chapter provides an overview of the core technologies and concepts relevant to this thesis. It introduces the JavaWiz system, explains the underlying principles of debugging and program visualization, and describes the GraalVM Espresso engine as the foundation for implementing the record and replay functionality.

2.1 JavaWiz

JavaWiz is a visual debugger developed at the Institute for System Software of the Johannes Kepler University Linz [1, 2]. It is designed with a strong emphasis on usability and educational effectiveness. Its usability is guided by the following principles:

- **Low entry barrier**: No complex installation or learning curve.
- **Comprehensiveness**: Covers key threshold concepts such as references, memory layout, and control flow.
- **Authenticity**: Avoids abstraction mismatches by showing real execution behavior.
- **Configurability**: Supports different teaching styles by allowing lecturers to tailor the interface based on the learning objectives.

By combining these features, JavaWiz facilitates active learning and supports students in developing accurate mental models of program behavior.

The backend of JavaWiz is responsible for executing Java applications and extracting detailed runtime information. It serves as the execution engine that collects traces and transfers data to the frontend. Currently, two distinct variants of the backend are implemented.

JDI Backend The first variant of the backend [1] is based on the Java Debug Interface (JDI) [6], a standard API provided by the Java Platform for interacting with the Java Virtual Machine (JVM) during program execution. This approach connects to a running JVM instance and utilizes the JVMs debugging capabilities to extract the needed runtime information. Although this setup is the standard way to debug Java environments, the JDI API itself is not designed for deep or flexible introspection and imposes constraints on how much and how precisely the execution can be monitored or modified, making it less suitable for advanced dynamic analysis or alternation of the control flow. Moreover, runtime overhead in using JDI is high.

Espresso Backend The second variant of the backend [5] is based on GraalVM Espresso [7], a Java bytecode interpreter implemented in Java that allows for more flexible introspection and modification of the execution process. With WebImage, the Espresso based backend can be compiled to JavaScript, allowing client-side execution within a browser without needing local Java installations. This architecture eliminates server-side code execution and improves scalability, accessibility, and security.

The backend processes Java source code through the following steps, illustrated in Figure 2.1:

- Code Instrumentation: Tracing methods are inserted directly into the source code within control structures and array accesses to monitor runtime behavior, ensuring no side effects are introduced.
- Abstract Syntax Tree (AST) Extraction: JavaWiz uses the parser of javac to generate ASTs, which serve as the foundation for the visualizations in the frontend.
- Compilation and Debugging: The modified code is compiled and executed in a debug mode using either the Java Debug Interface (JDI) or the Truffle Debug API (in the web version via Espresso VM).
- Execution Tracing: During execution, JavaWiz builds and handles "trace states" that include the call stack, heap, variable states, standard I/O activity, and control flow information.

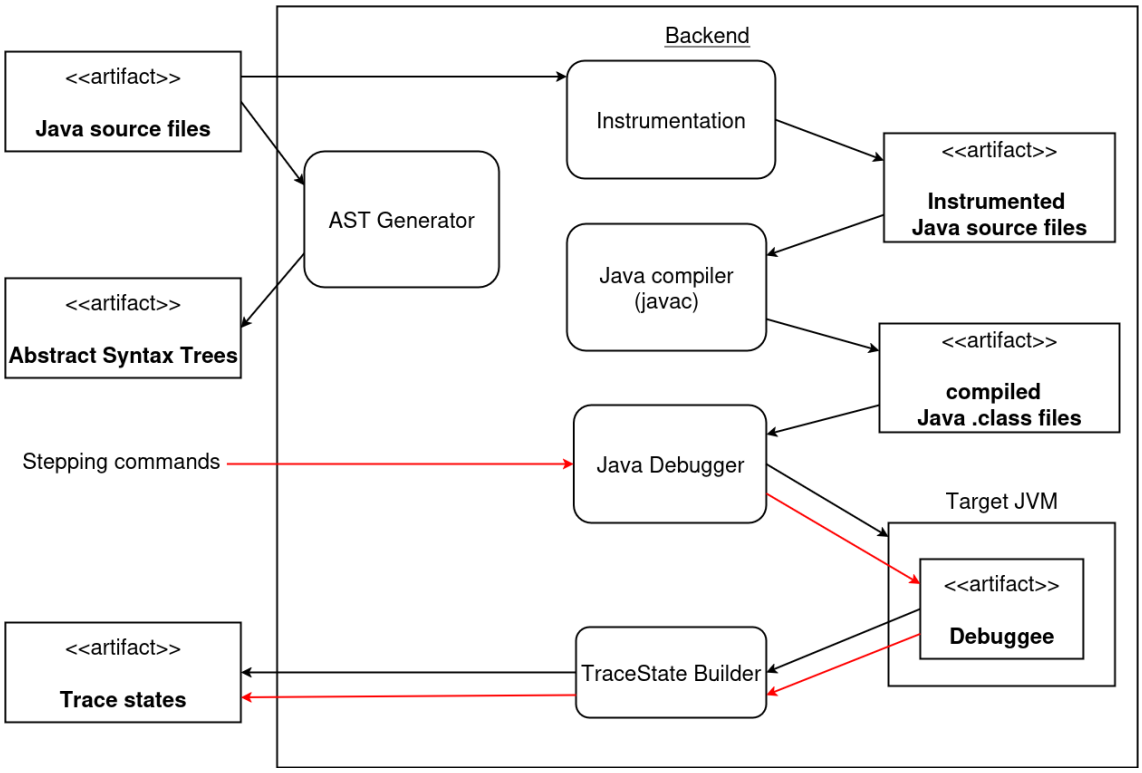


Figure 2.1: Overview of the JavaWiz backend processing steps. Black arrows indicate processing steps during compilation and initialization. Red arrows indicate processing steps during execution.

JavaWiz Frontend The frontend of the JavaWiz debugger offers, unlike conventional debuggers, multiple visualizations on program execution, including:

- Flowcharts based on control flow from ASTs.
- A memory view of the heap and stack, including primitive values, references, and static fields.
- A desk test visualization that maps variable states to executed lines.
- Specialized views for arrays, linked lists, and binary trees.
- Input/output monitoring with buffer inspection.

These interactive data driven visualized views are integrated into the web-based frontend. Figure 2.2 shows a screenshot of the JavaWiz execution. The current execution state of a `BookExample` program, that stores a title and page count in a `Book` object and prints the title to the console, is shown in four views as follows:

(1) **Stack and Heap:** The stack shows the current method call `BookTest.main(...)` with the local variable `title` referencing the value `"Java"`. The heap contains a `Book` object referenced by the static variable `b`, which stores a `String` object with the value `"Java"` and an `int` value `500` in its `pages` field. The heap also contains the `String[] args`, which is currently empty.

(2) **Desk Test:** The desk test panel lists the instructions executed in sequence: creation of the `Book` object and assignment to the static variable `b`; invocation of the method `getTitle()` on `b`; retrieval of the value `"Java"` from the field `title`; and assignment to the local variable `title`.

(3) **Flow Chart:** The flow chart shows the sequence in the `main()` method: call `b.getTitle()`, store the result in `title`, and print `"Title: Java"` to the console.

(4) **Sequence Diagram:** The sequence diagram depicts `main()` calling the method `getTitle()` on the `Book` object, retrieving the value of its `title` field, and returning it to `main()` for output.

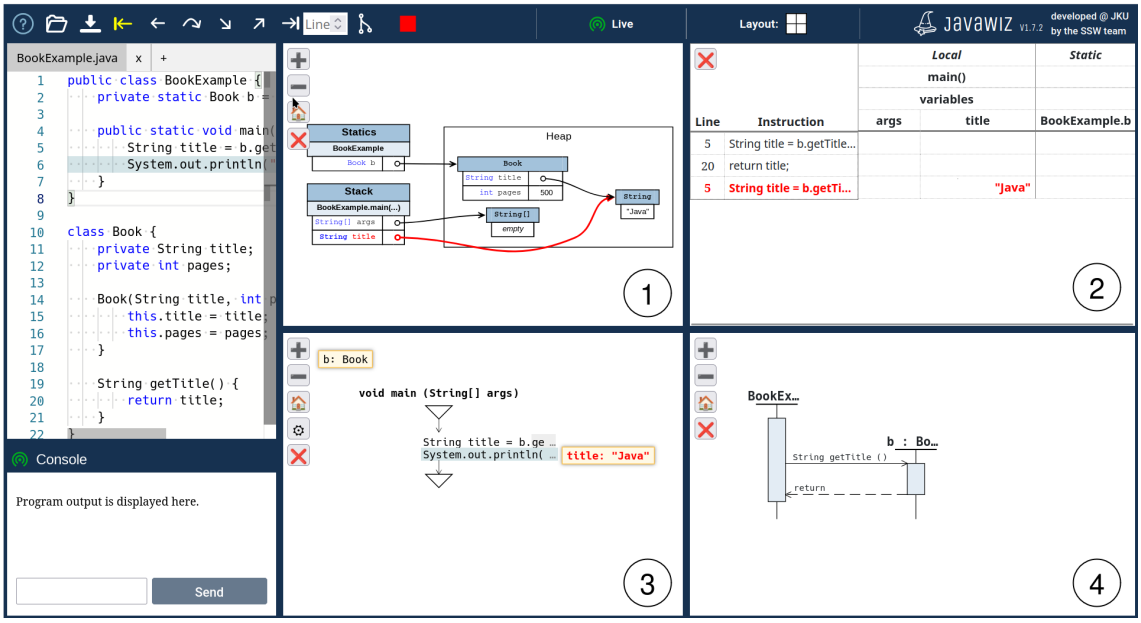


Figure 2.2: Screenshot of JavaWiz execution.

2.2 GraalVM

GraalVM is a high performance polyglot virtual machine developed by Oracle Labs [8]. It is designed to run applications written in multiple programming languages such as Java, JavaScript, Python, Ruby, and R within a shared runtime environment [9] and is built on top of the standard Java HotSpot VM (Figure 2.3). GraalVM is distinguished by its support for interoperability between languages and its ability to optimize programs across language boundaries through a common execution platform [10].

In the context of Java, GraalVM introduces several innovations over traditional Java Virtual Machines, including support for ahead-of-time (AOT) compilation via native images, a just-in-time (JIT) compiler written in Java, and full integration with the Truffle framework [11]. These features make GraalVM not only a high performance platform for production environments but also a flexible and introspective platform for research and educational purposes.

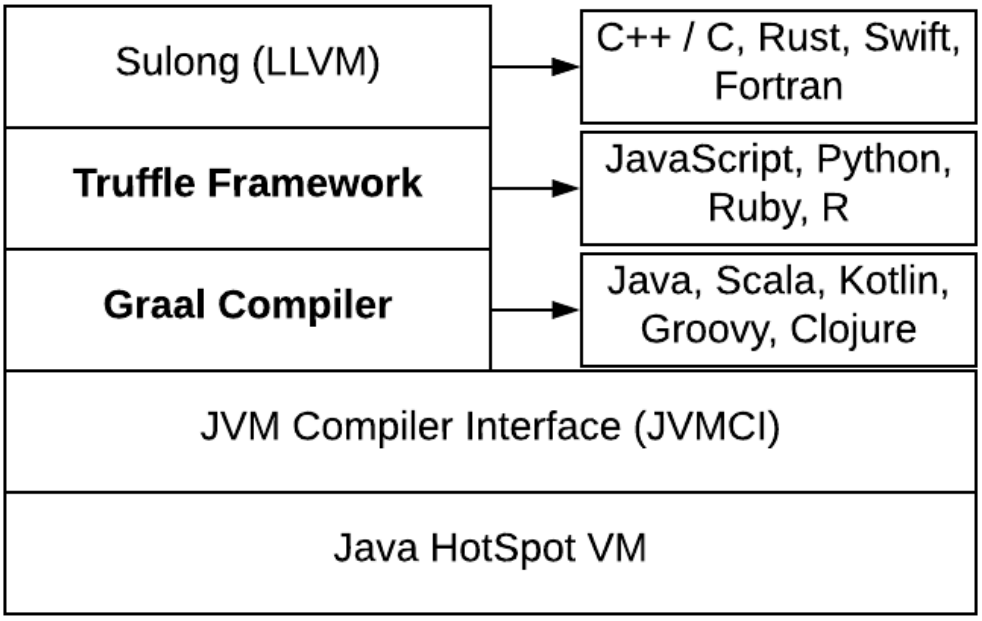


Figure 2.3: Overview of the GraalVM architecture [12].

This thesis focuses on leveraging the interpretive infrastructure of the GraalVM to develop a capture and replay debugging backend for JavaWiz. The next subsections outline the Truffle language implementation framework and the Espresso interpreter, both of which play a central role in the realization of this work.

2.2.1 Truffle

The Truffle framework is a language implementation framework designed for building abstract syntax tree based interpreters for dynamic and statically typed languages [13, 14]. It is a core component of GraalVM and provides the tooling necessary for writing language interpreters that are both efficient and highly instrumentable.

Truffle-based interpreters execute programs by walking their syntax trees and can be dynamically optimized [10] at runtime using the JIT compiler of the GraalVM. However, what makes Truffle particularly relevant for this thesis is its support for instrumentation, introspection, and execution control.

These capabilities include:

- Node instrumentation: Each node in the AST can be observed, modified, or wrapped to monitor execution behavior.
- Execution events: Hooks can be attached to function calls, variable reads/writes, and control flow changes.
- Custom debuggers and profilers: Truffle provides APIs for building tools that inspect and influence the runtime behavior of guest applications.

These features make the Truffle framework an ideal foundation for implementing recording mechanisms and execution replays, as it allows fine-grained control over what happens during interpretation without modifying the application code itself.

2.2.2 Espresso

Espresso (Java on Truffle) is a Java bytecode interpreter implemented in Java on top of the Truffle framework [7] (Figure 2.4). As part of GraalVM, Espresso executes `.class` files by interpreting their bytecode directly, rather than compiling it to native code. This interpretive approach aligns well with the needs of tool developers, language researchers, and educators.

The key properties of Espresso relevant to this thesis include:

- Modularity: Espresso is entirely written in Java, making it easier to extend and modify compared to native JVM implementations [16].
- Introspectability: As a Truffle-based interpreter, Espresso exposes its internal execution model, allowing external tools to observe and manipulate runtime state [13].
- Execution hooks: Espresso supports the interception of standard library methods and native interactions, which is critical for recording nondeterministic behavior (e.g. time queries, I/O operations) [17].

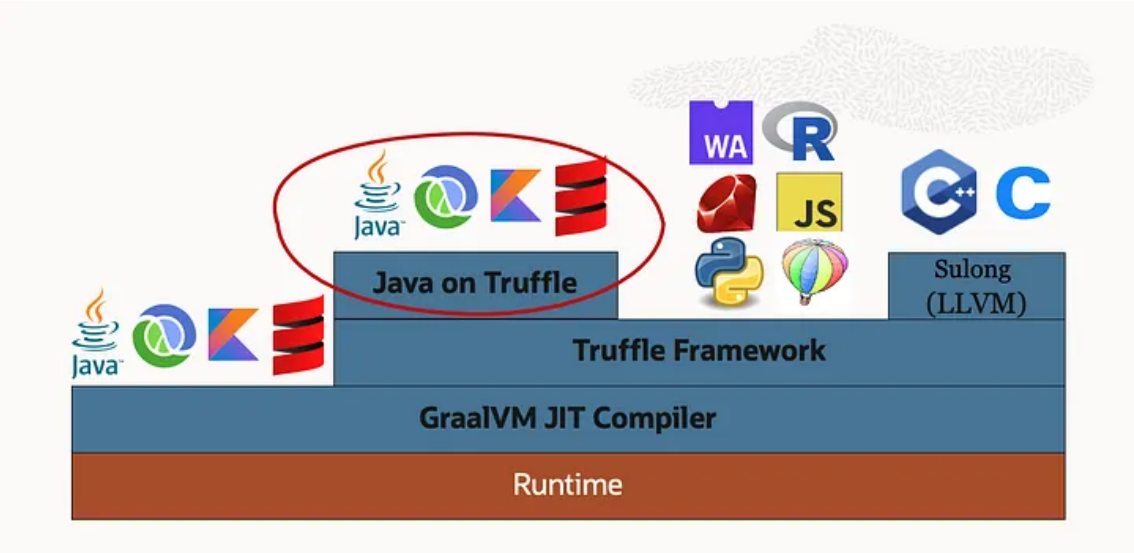


Figure 2.4: Java on Truffle in the GraalVM architecture [15].

These characteristics position Espresso as a suitable platform for building a Record and Replay system. In particular, its ability to replace native method calls with custom logic enables the capture and replay of external inputs, which is central to this thesis.

By extending Espresso with custom instrumentation, it becomes possible to intercept method invocations, store relevant execution data, and later reproduce the same application behavior. Furthermore, the introspective capabilities of Espresso provide a foundation for implementing state snapshots, enabling the restoration of application state at specific points in time.

3 Capture and Replay Approach

This chapter outlines the foundations for the implementation of the capture and replay functionality in the context of Java programs in JavaWiz. It defines the fundamental components of a recording and replaying system, highlights technical challenges specific to the Java ecosystem, and motivates the selected approach based on GraalVM Espresso and the Truffle framework with educational use cases.

3.1 Overview

The foundation of the capture and replay functionality presented in this thesis lies in the concept of Deterministic Replay Debugging (DRD) [18, 19]. This debugging paradigm enables an exact reproduction of the execution of a program, including its control flow, variable states, and interactions with the environment. DRD addresses a key limitation of traditional debugging, the inability to reliably reproduce nondeterministic behaviors such as concurrency bugs, race conditions, or input-dependent anomalies.

Unlike conventional debugging methods where re-executing a program may yield different behavior due to mentioned nondeterministic factors, DRD guarantees that a replayed execution behaves identically to the original run. This is achieved by recording all sources of nondeterminism during the program execution and reinjecting the recorded data during replay. Examples of such sources include user inputs, system time dependent calls, and native method invocations.

Our approach in this thesis is to adapt the DRD concept to work within JavaWiz for the capture and replay mechanism, using GraalVM Espresso and the Truffle framework. These technologies allow for deep introspection of program execution, which are essential for achieving full determinism in replay scenarios of executed Java programs.

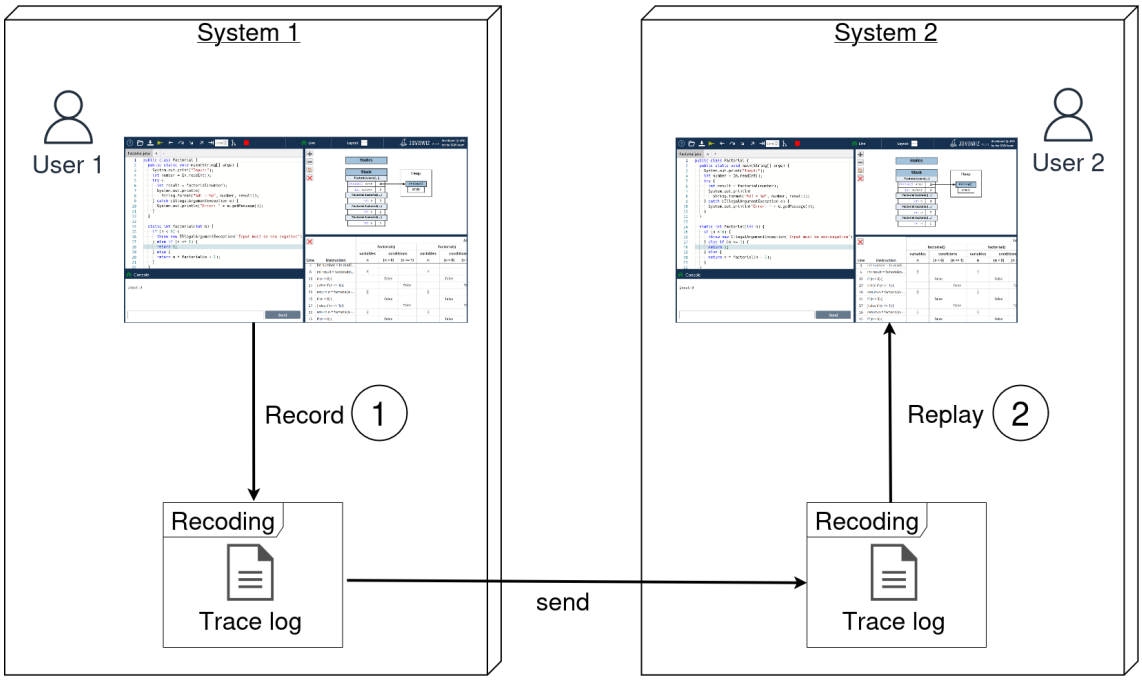


Figure 3.1: Deterministic Replay Debugging. The left side shows the recording phase (System 1), where the program is executed in JavaWiz and nondeterministic events are captured. The right side shows the replay phase (System 2), where the recorded data is used to reproduce the original execution behavior.

The proposed capture and replay mechanism in JavaWiz is based on the following two-phase model (shown in Figure 3.1):

(1) Recording Phase: During normal execution, the system intercepts and stores non-deterministic events and inputs that influence the program behavior. This includes system calls (native calls), user interactions, and environmental factors.

(2) Replay Phase: The program is re-executed in a deterministic manner, simulating the original execution environment using the previously recorded data. All nondeterministic events are replaced with their corresponding recorded values to ensure identical behavior up to the latest recorded state, after that the program continues to run normally. The re-execution can be run either on the same system or on a different system.

By leveraging this approach, JavaWiz gains the ability to reproduce and inspect program runs in a deterministic manner. This functionality not only benefits debugging and analysis

tasks, but also introduces educational opportunities, such as replaying sessions prepared by lecturers, comparing alternative executions, or resuming from saved snapshots in classroom settings. The result is a fully reproducible execution that allows students and developers to inspect and debug the program live.

3.2 Design Considerations for Java Replayability

To adapt DRD in JavaWiz, it is essential to account for Java-specific runtime behaviors that influence the reproducibility of program execution. A central challenge arises from calls of native methods within the Java Virtual Machine, which serve as a bridge between platform-independent Java bytecode running on the JVM and platform-dependent native code such as C or C++ [20].

Native methods are invoked directly by the JVM through the Java Native Interface (JNI) [20] and usually involve operations that depend on the underlying operating system or hardware state. Because these methods are not run in the JVM, their behavior is not transparent and can not be controlled within Java code, thus they introduce sources of nondeterminism into the execution of the Java program.

The most important Java operations based on native methods that we need to consider for capture and replay in JavaWiz are input and output handling, such as reading from `System.in`. Similarly, calls to environment variables, e.g. `System.getenv()`, rely on system state that may vary between executions. Even seemingly straightforward operations like obtaining the current time with `System.currentTimeMillis()` or `System.nanoTime()` may involve non-repeatable behavior if not properly controlled.

These factors underscore the importance of a capture and replay model that can intercept and record native interactions at a fine-grained level. Therefore, our approach must ensure that nondeterministic effects, originating from native code, are recorded during the initial run and faithfully reproduced during replay. This is fundamental to supporting reliable and meaningful re-executions of Java programs in context of our educational use cases, outlined in Section 4.3.

3.3 Role of Truffle and GraalVM Espresso

Based on the implementation of the Espresso backend variant, mentioned in Section 2.1, we now have the opportunity to instrument and intercept Java programs with high flexibility provided by the Truffle framework and GraalVM Espresso.

- Truffle provides a framework for building interpreters with full control over the execution semantics [13]. It allows us to intercept execution at the language level (e.g. Java on Truffle), which is essential for building a deterministic replay mechanism.
- The Espresso VM implemented in Java allows high-level introspection without the restrictions of standard JVMs like HotSpot. This makes it suitable for implementing interceptors at a high level to observe nondeterministic behavior during execution.

This toolchain enables a form of language-level virtualization in which nondeterministic operations can be intercepted, logged, and later replaced with deterministic values without requiring bytecode rewriting or modifying the standard JVM. Furthermore, we will utilize GraalVM Web Image to compile the Espresso backend to JavaScript.

3.4 Event Recording and Replay

To replay a Java program exactly as it ran the first time, we need to record important details about what happened during the original execution. Certain operations can behave differently each time the program runs, such as reading the current time, asking the user for input, or generating random numbers.

To ensure that replay faithfully reproduces the original execution, the system must preserve the exact order of events and avoid re-invoking the original operations. Instead, it should rely solely on the recorded data to reconstruct the program behavior.

3.4.1 Why Recording is Necessary

Even in simple programs, nondeterministic operations can cause variations in behavior. Consider the following code:

```
System.out.println(System.currentTimeMillis());
System.out.println(Math.random());
```

This prints the system time and a random number. If we run this twice, the produced output will differ:

- Run 1:

  ```
  1721224876221
  0.845392123
  ```

- Run 2:

  ```
  1721224990320
  0.152290339
  ```

To make the replay output identical to the original run, we record the values produced by these operations and reuse them during replay.

3.4.2 Handling Nondeterministic Operations

To ensure replicability, the original run, we log the results of these operations. During replay, instead of executing the original operation again, we return the saved value and bypass the original source of variability.

For example:

- `System.currentTimeMillis()` → `1721224876221`
- `Math.random()` → `0.845392123`
- `In.readLine()` → `"lena"`
- `System.getenv("HOME")` → `"/home/lena"`

In the next example we illustrate how JavaWiz shall handle interactive user input during replay. Figure 3.2 shows the execution of a simple program that prompts the user for their name and prints a personalized greeting. Although the interaction appears straightforward, it involves a nondeterministic operation that waits for user input, which must be recorded to ensure consistent behavior during replay.

```
1 System.out.println("Enter your name:");
2 String name = In.readLine(); // user types "Lena"
3 System.out.println("Hello, " + name);
```

The capture system records the original user input `"Lena"`. During replay, this value is automatically returned, and the user is not prompted again. This guarantees that the replayed output (`"Hello, Lena"`) matches the original execution exactly.

Figure 3.2 visualizes this replay state in JavaWiz:

1. The Code View shows the program started as a replay at the `System.out.println` statement on line 3.
2. The Execution Tree View highlights the current execution step, and the tooltip for the later step at line 5 shows the recorded input `"Lena"`.
3. The Stack and Heap View displays the `main` stack frame with the `args` parameter pointing to an empty `String[]` in the heap. Since the `name` variable has not yet been assigned, the heap currently contains no additional objects.
4. The Flow Chart View shows the sequence of instructions in `main()`, with the first statement highlighted.

By preserving the results of all unpredictable operations and respecting their order, JavaWiz ensures deterministic replay. This makes program behavior reproducible and predictable, which is critical for reproducible analysis, effective teaching workflows, and collaborative analysis.

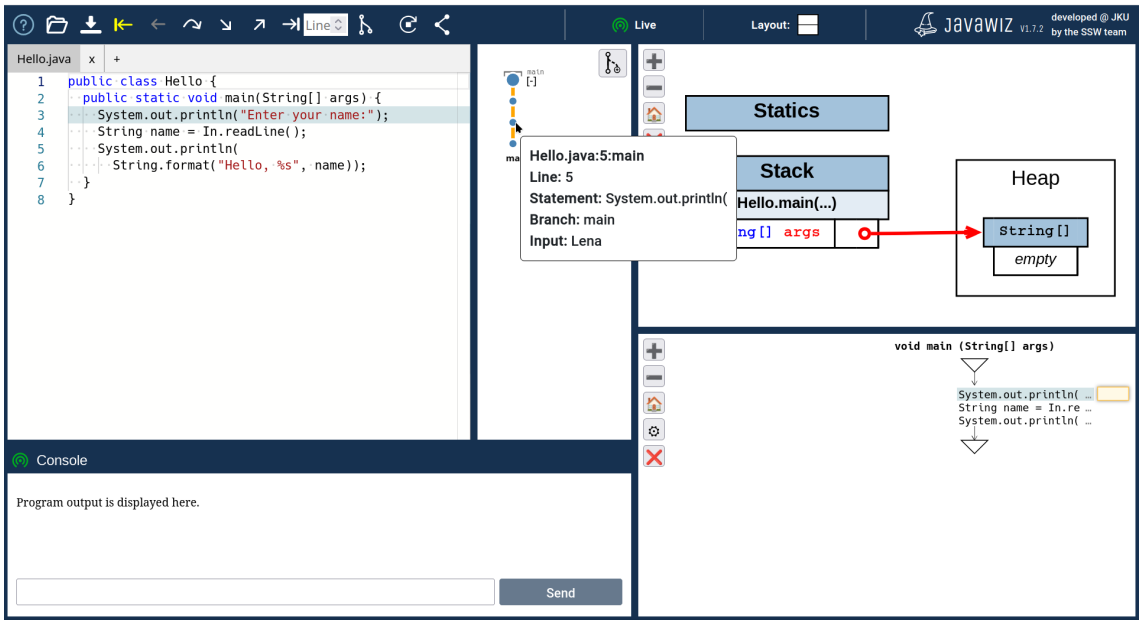


Figure 3.2: Execution of a simple input/output program in JavaWiz paused at line 3. At this point, the heap contains only the empty `String[] args`, and recorded user input (`"Lena"`) will be replayed later without re-prompting.

4 User Interface and Use Cases

This chapter describes the user interface design and interaction features of the JavaWiz application that supports recording, replaying, and sharing of Java program executions. We highlight how users can interact with the system to capture execution traces, explore program behavior, and share insights with others. The chapter also presents practical use cases that demonstrate the educational value of these features in teaching programming concepts.

4.1 User Interaction

In this section, we describe how end users interact with the JavaWiz application that supports recording, replaying, and sharing of Java program executions. We demonstrate key features with visual examples and provide an overview of the system workflow.

Features

The following features are integrated into the user interface and are available to the user:

- Recording: Enables the capture of execution data of nondeterministic events (e.g. user inputs).
- Branching: Allows users to create alternative execution paths from a specific point in the recorded trace, enabling exploration of different scenarios with different inputs.
- Replaying: Allows the user to step through a previously recorded session, replicating the original behavior, resume execution and explore different execution scenarios by creating branches.

- Sharing: Recordings can be shared via URLs for collaborative debugging or instruction.

System Workflow

The following steps outline the updated user interaction flow that includes our capture and replay feature in JavaWiz:

(1) The user initiates a debugging session with the recording feature enabled (Figure 4.1).

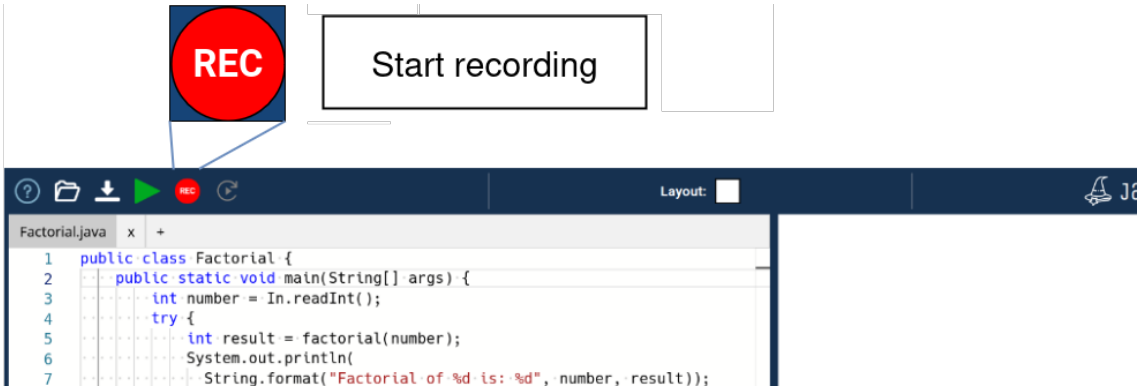


Figure 4.1: The user can start the recording from the toolbar.

(2) The debugging session starts and the execution is visualized (Figure 4.2). The user can execute the program in a step-wise manner and investigate execution states using different visualizations. The execution tree evolves as the debugging processes (Figure 4.3). Users are also able to navigate through the execution trace and inspect program states at any point in time. It is possible to toggle the visibility of the trace tree at any time.

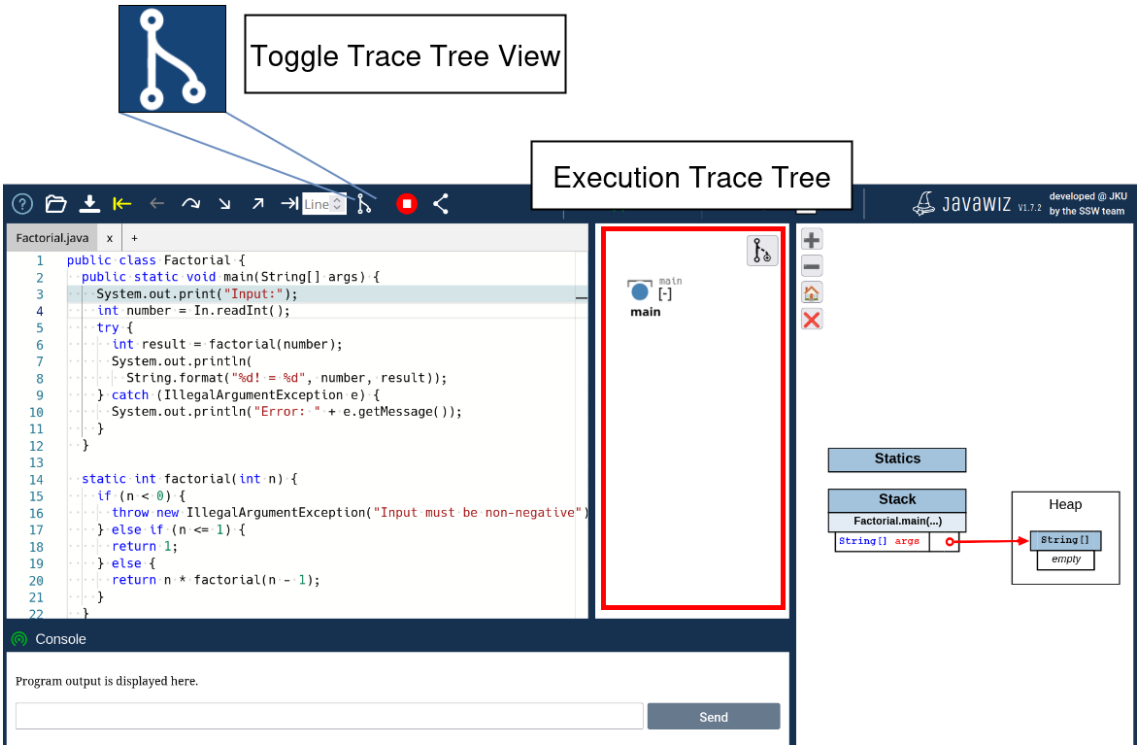


Figure 4.2: Toggled Execution trace tree showing the initial program state.

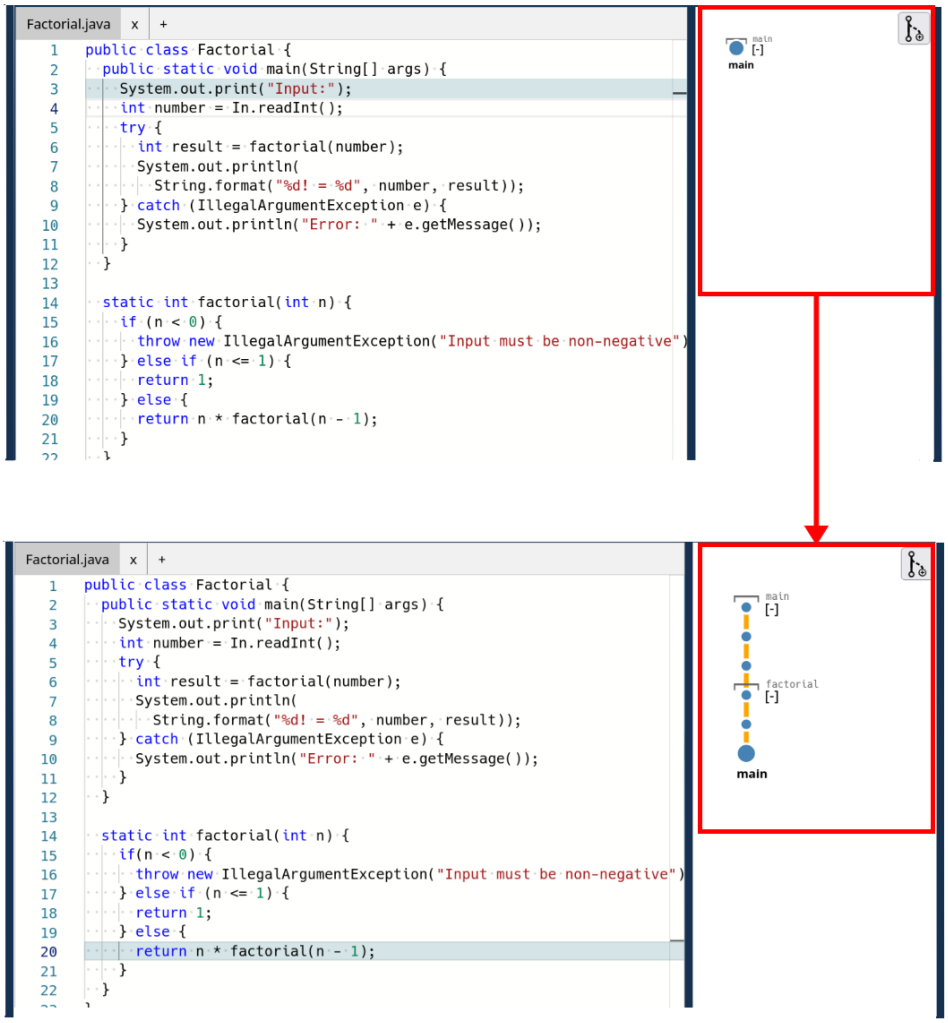
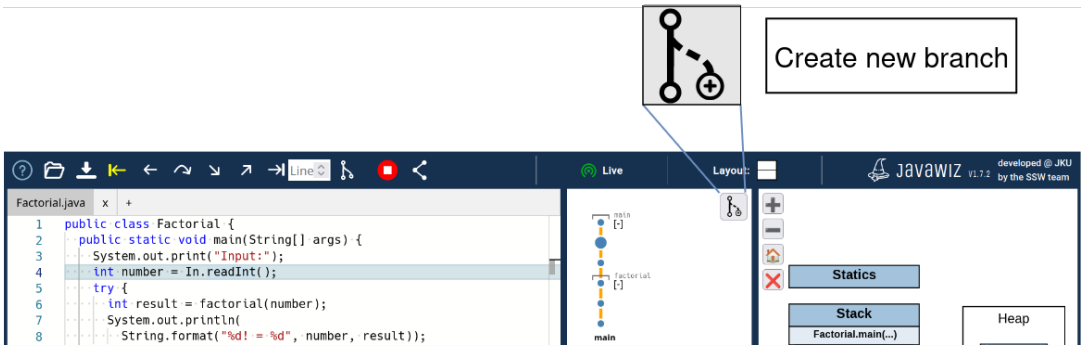
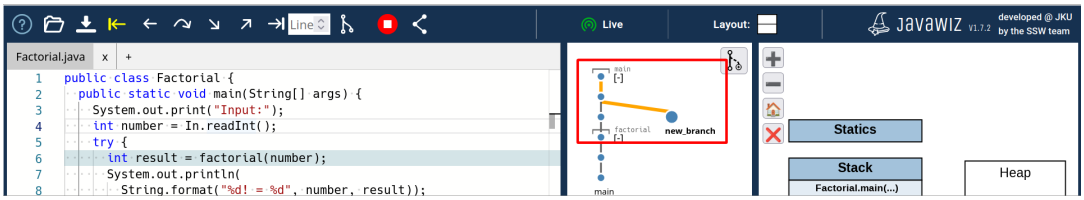


Figure 4.3: Execution trace tree evolves.

③ During execution, the user has the option to create a branch at any point in the execution trace. This allows them to explore alternative paths, compare execution scenarios, or test different inputs seamlessly (Figure 4.4).



(a) New branches can be added interactively from the trace tree.



(b) New branch is created with a new trace state.

Figure 4.4: Branching from a trace state.

(4) Users can pause execution at any time for sharing a trace of the captured program, including its states for later replay and resumption (Figure 4.5).

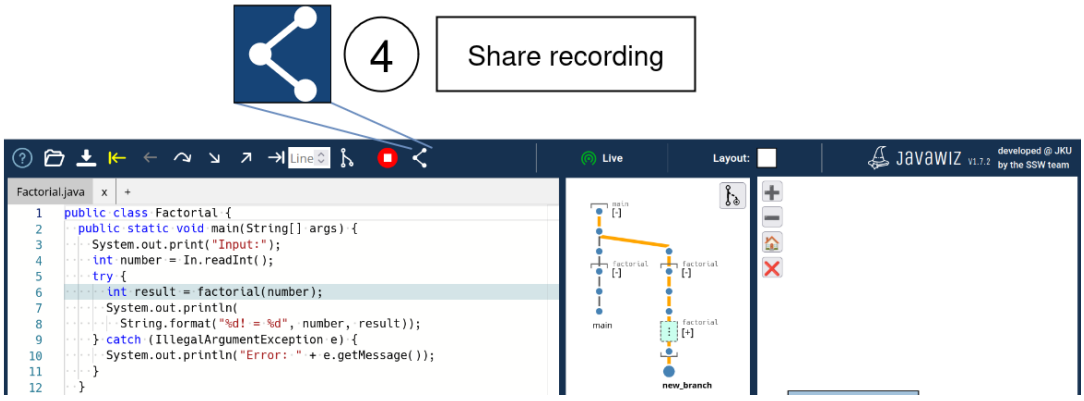


Figure 4.5: Snapshots can be saved using share option from the toolbar.

(5) A previously recorded execution can be loaded by opening a shared link. Once a previously recorded session is loaded, users are presented with the full execution context, including the original source files, execution trace, and program state history (Figure 4.6).

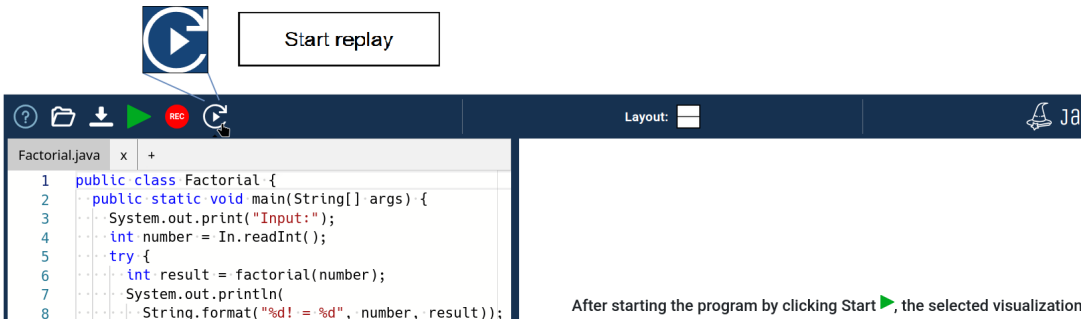


Figure 4.6: The user can start the replay from the toolbar.

The replay interface allows users to step through the program execution, moving forward or backward across statements (Figure 4.7). Each step updates all relevant visualizations, including the call stack, heap, and execution trace deterministically, so that the user can inspect how the program behaved on the original recorded execution. This navigation is particularly useful for replicating bugs or tracing the impact of a specific input.

In addition to stepping, users can freely navigate through the trace tree, which provides a structural overview of control flow and method calls. The trace tree not only shows the linear sequence of executed statements, but also branches and method calls, allowing for quick jumps. At every node in the trace, the system reveals detailed data. Users can view detailed information about each execution state, such as source file name, line number, executed statement, and branch name.

The system supports branching from any trace state. A user can return to a point in the trace tree, create a branch, introduce a different input, and continue the program along a new path. This branching model is ideal for comparative analysis, such as testing different execution scenarios.

Moreover, if a run of a debugging session has not been completed during recording, JavaWiz ensures continuity by preserving the entire trace and state history. Users may pick up exactly where the execution was left off.

Together, these features offer a model for replay-based program analysis for JavaWiz to understand execution behavior in depth.

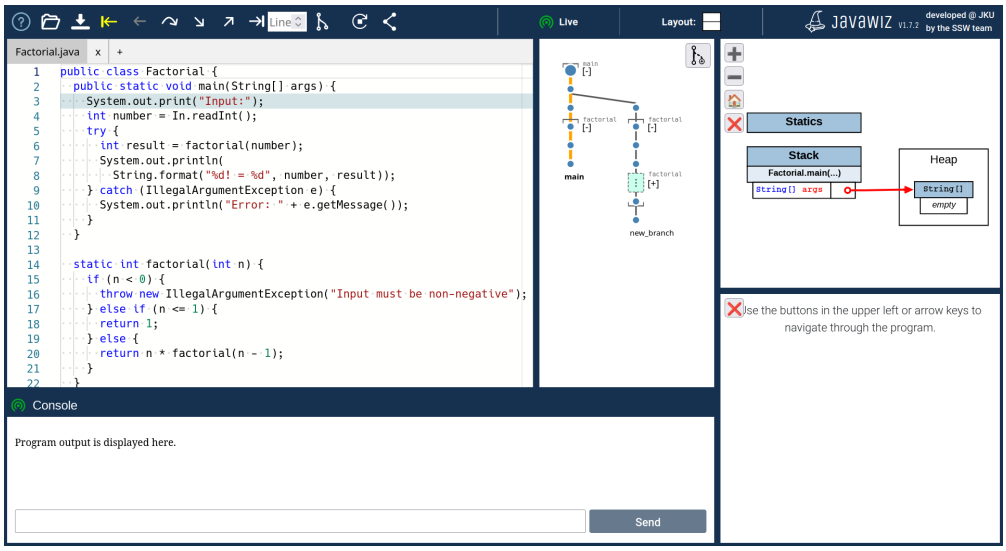


Figure 4.7: Loaded replay session with execution trace tree.

4.2 Visual Design and Interaction Features

In this section, we present the specific features of the capture and replay system in JavaWiz.

4.2.1 Program execution trace tree

The program execution trace tree in JavaWiz offers an interactive visualization of the program execution flow. Displayed as a hierarchical, top-down tree layout (shown in Figure 4.8), it enables users to explore method calls, control flow, and branching structures in a clear and comprehensible manner. Each node in the tree represents a distinct program state corresponding to a specific source line within the given program.

In this view, the vertical axis represents the temporal progression of the program, while the horizontal axis reflects logical branching. Branches are spaced to maintain readability, ensuring a clear distinction between separate branches even for more complex traces. The following visual elements support user navigation and understanding:

(1) Circle markers denote individual program states within the trace.

(2) An enlarged highlighted node indicates the current active execution state, allowing users to track progress in real time or during replay.

(3) Rectangular boxes with ellipsis (⋮) represent collapsed method calls, summarizing larger execution segments to reduce visual noise. Those boxes can be expanded to show internal execution nodes.

(4) The entry and exit of the method are illustrated by brackets (⌐ and ⌙), with the method name displayed adjacent for quick orientation. The method entry points provide toggles ([+] and [-]) to expand or collapse their internal trace, giving users flexibility when navigating complex call structures.

(5) Branch labels beneath each node clarify branch associations and improve interpretability.

Users may interact with the trace tree in multiple ways. Clicking on a node allows stepping directly back to that specific program state, while hovering reveals a tooltip containing key metadata such as the file name, line number, executed statement, and branch identifier.

To maintain orientation, the currently active execution node is highlighted as a larger node. The entire path along the active branch as well as the branch name are highlighted, providing a clear sense of control flow and progression.

By combining these graphical cues with interactive navigation, the program execution trace tree offers an accessible yet powerful means of exploring even complex Java program behaviors within JavaWiz.

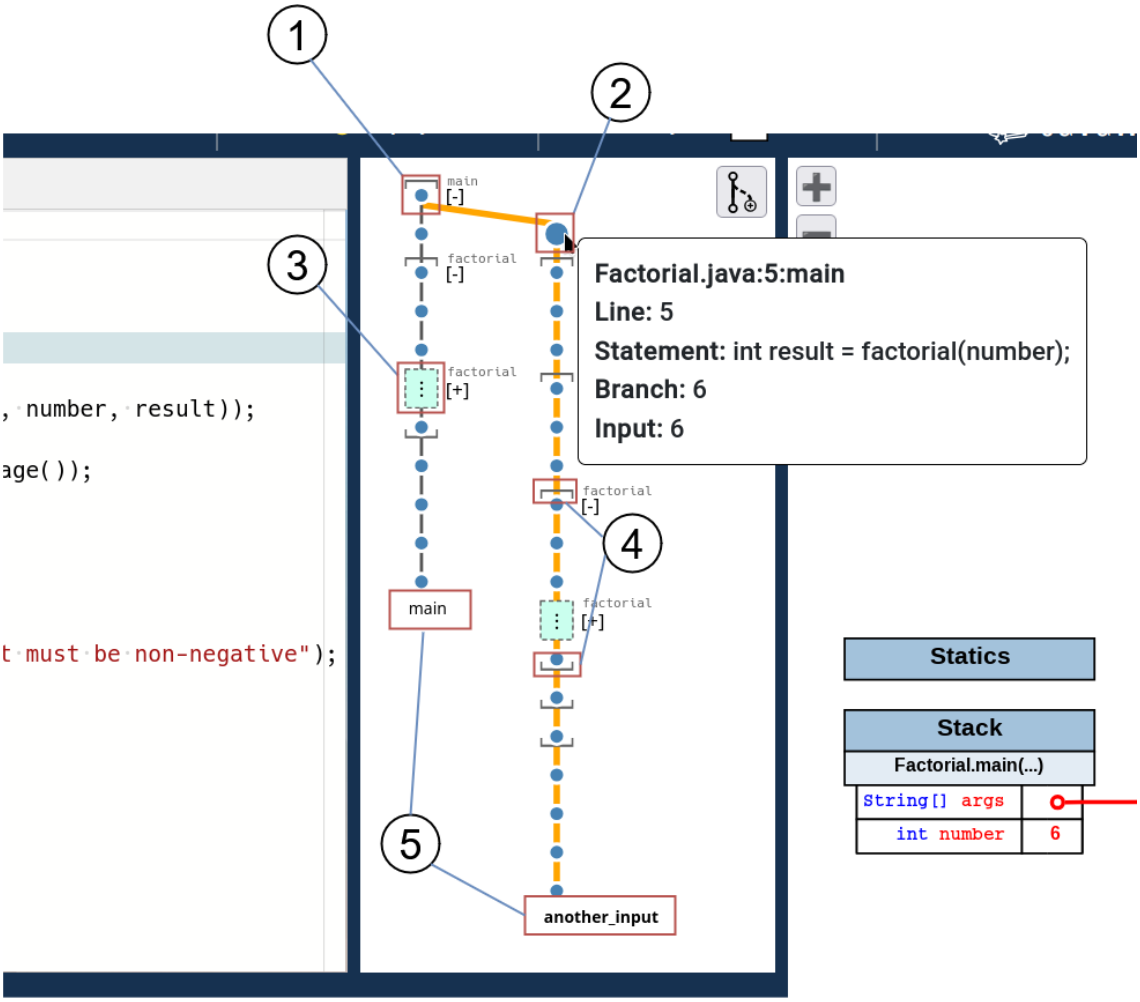


Figure 4.8: Interactive program execution trace tree in JavaWiz.

4.2.2 Sharing

The sharing functionality in JavaWiz allows users to export recorded execution traces, including all relevant metadata, as a shareable URL. This feature is particularly useful for collaborative debugging, educational purposes, or sharing insights with peers or instructors.

When a user records an execution trace, they can generate a unique URL that encapsulates the entire trace data.

Figure 4.9: Popup modal dialog for sharing in JavaWiz.

The generated URL is shown in a modal dialog (see Figure 4.9). This URL can be shared via email, messaging platforms, or embedded in educational materials. Recipients of the URL can then load the trace directly into their own instance of JavaWiz, allowing them to replay the execution, inspect program states, and explore the control flow without needing access to the original runtime environment.

4.3 Examples of Educational Use Cases

In this section, we present potential scenarios of the capture and replay functionality in JavaWiz, focusing on how it can enhance the learning experience for students studying programming concepts.

4.3.1 Scenario 1: Teaching Exception Handling

Lecturers can introduce the concept of exceptions in an introductory programming course using a simple numeric input parser (Listing 4.1). To highlight both valid and invalid behaviors, the instructor prepares two executions in JavaWiz (shown in Figure 4.10).

In the first recording, the program is executed with an invalid input such as `"notANumber"`, which causes `Integer.decode()` to throw a `NumberFormatException`. The JavaWiz trace clearly shows where the exception is raised and how it propagates through the method calls. Students can observe the failed parsing step and the exact execution point at which the error occurred.

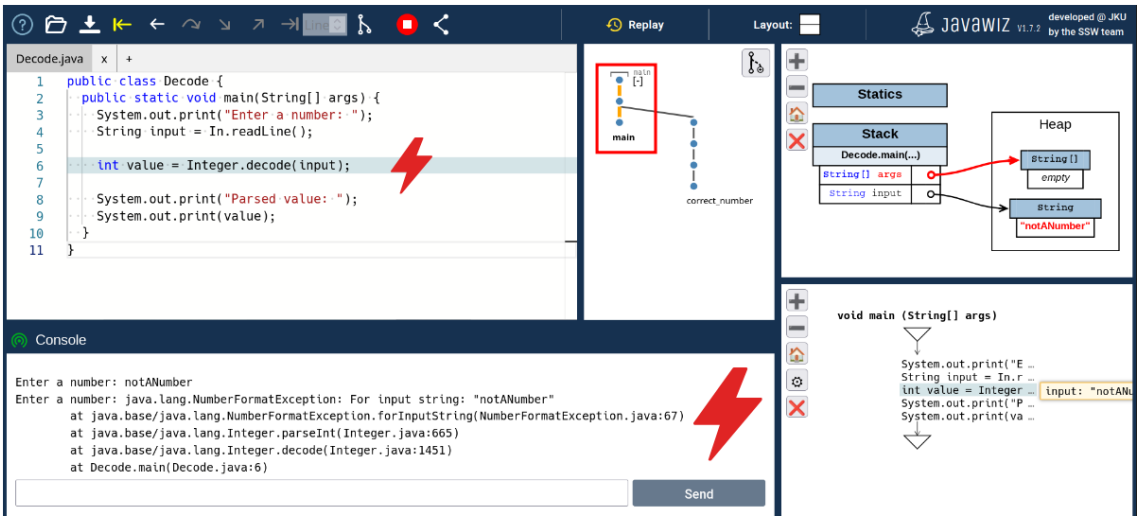
In the second recording, the input provided is valid (e.g. `"42"`), resulting in successful parsing and display of the output. The visualization shows how the input string is converted to an integer and passed through the method flow without interruption.

The students are then given the task of replaying both recordings. They can inspect the trace tree to locate the exception node, compare it with the successful execution path, and examine the corresponding call stack and heap state.

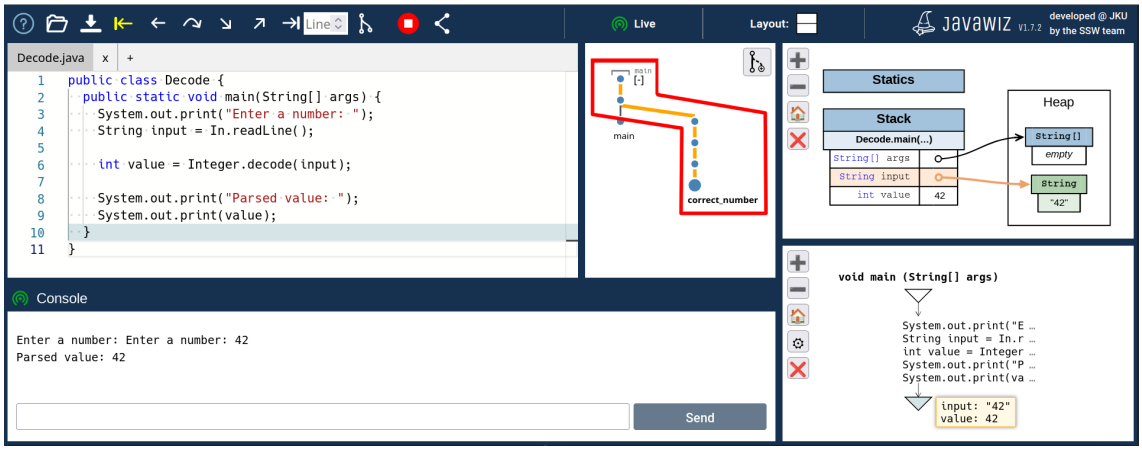
This scenario allows students to understand the mechanics of exception handling and the structure of input-driven errors in the program control flow.

```
1  public class Decode {
2      public static void main(String[] args) {
3          System.out.print("Enter a number: ");
4          String input = In.readLine();
5
6          int value = Integer.decode(input);
7
8          System.out.print("Parsed value: ");
9          System.out.print(value);
10     }
11 }
```

Listing 4.1: Example of user input parsing with `Integer.decode()`



(a) Execution with invalid input (`"notANumber"`) showing `NumberFormatException` being thrown in the `main` branch.



(b) Execution with valid input (`"42"`) showing successful parsing and uninterrupted control flow in the `correct_number` branch.

Figure 4.10: Screenshot of JavaWiz Replay Scenario 1.

4.3.2 Scenario 2: Exploring Input-Driven Behavior through Replay Branching

When teaching user input parsing and number representation, it is valuable for students to explore how different input formats affect the interpretation of numeric values in programming languages. The replay functionality in JavaWiz enables students to explore these variations without modifying the source code or rerunning the program from the beginning.

Using the same program provided previously (Listing 4.1) as an example for this scenario, the lecturer can provide a JavaWiz recording that uses `Integer.decode()` to parse user input. The original execution uses the input `"42"`, which is interpreted as a decimal value and printed directly.

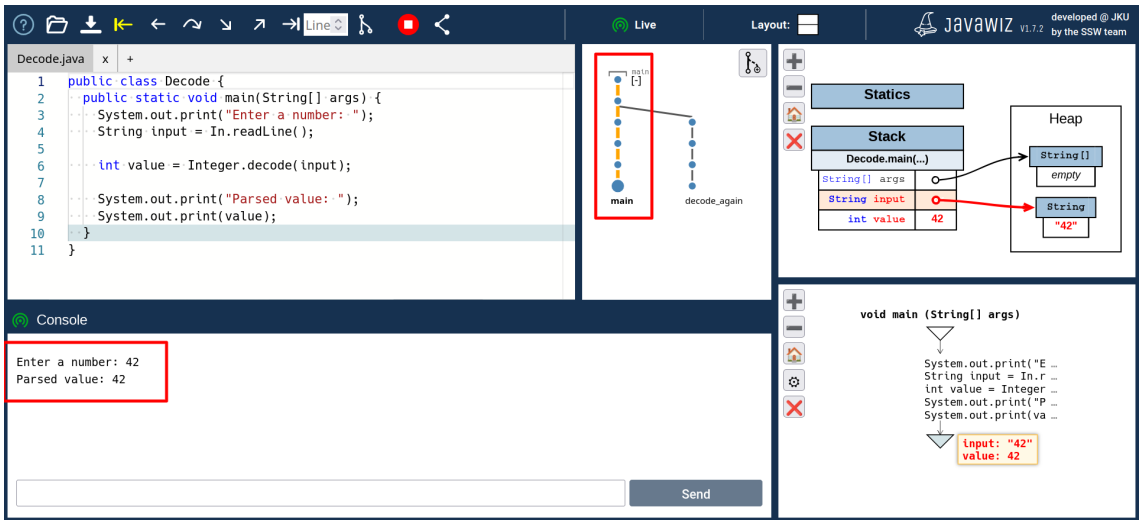
Students are then encouraged to explore how slight variations in input change the behavior of the program. For example, inputting `"042"` instead of `"42"` causes the method to interpret the value as octal representation, resulting in the decimal value `34`, as shown in Figure 4.11. This comparison highlights how leading zeros in numeric strings can introduce subtle semantic changes in Java.

Using replay branching, students can branch the execution at the input prompt and try further inputs:

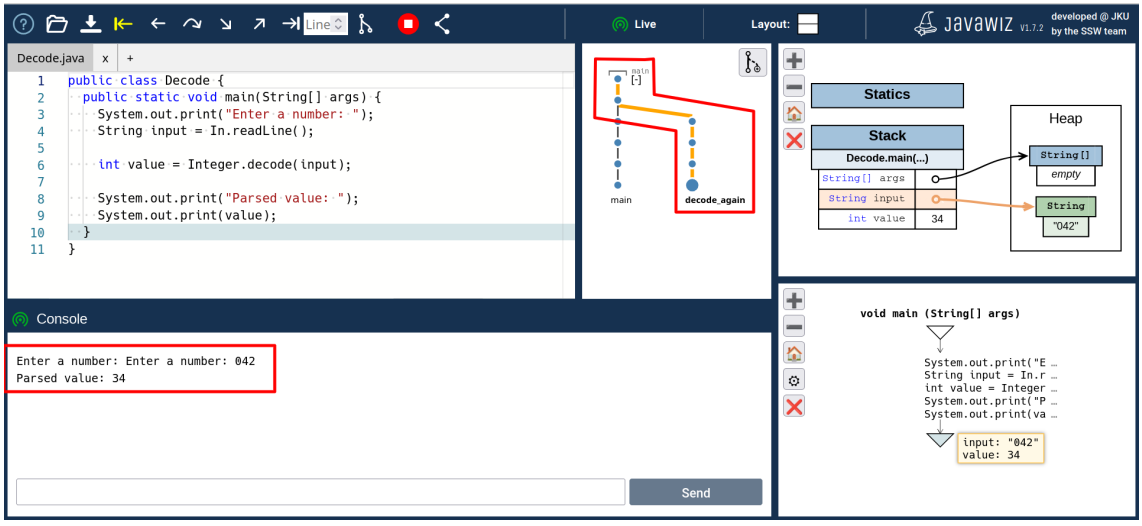
- `"0x2A"` - the hexadecimal representation of 42
- `"052"` - the octal representation of 42

The produced outputs are all observable through the JavaWiz interface. Students can examine the parsed result, observe any exceptions raised, and compare stack and heap states across the various branches.

This scenario helps students develop an intuition for how Java interprets numeric strings and how seemingly small input changes affect program control flow. The ability in JavaWiz to replay, branch, and visualize these execution paths supports exploratory learning and precise analysis.



(a) Original execution with input `"42"` interpreted as decimal and printed as 42.



(b) Branched execution with input `"042"` interpreted as octal, producing decimal output 34.

Figure 4.11: Screenshot of JavaWiz Replay Scenario 2.

4.3.3 Scenario 3: Trace Sharing and Remote Instruction

In distance or hybrid learning environments, students often struggle with debugging code without access to real-time instructor help. The capture and replay functionality in JavaWiz mitigates this challenge by enabling asynchronous guidance through shareable trace artifacts.

We consider a student working remotely on a recursive algorithm, e.g. computing permutations of a list (Listing 4.2). Upon encountering unexpected behavior or a stack overflow, the student captures the execution with JavaWiz and shares the link with the lecturer.

The instructor, instead of attempting to reproduce the issue from code alone, loads the execution trace directly. They can step through the trace, observe the recursive pattern, identify where and why the stack overflows, and create either a fixed execution as a branch or a new recording. The fixed version is then sent back to the student, who can replay the execution at their own pace.

Students benefit from seeing not only what was wrong, but also can compare the program execution versions. This feedback loop supports deeper learning and saves valuable time for both parties, as there is no need to reproduce the bug manually in a development environment.

This scenario highlights how JavaWiz enables asynchronous communication in software development education, turning debugging into a shared learning activity without the setup of development environments.

```
1  import java.util.*;
2
3  public class PermBug {
4      public static void main(String[] args) {
5          List<String> out = new ArrayList<>();
6          char[] a = {'A','B','C'};
7          permute(a, 0, out);          // student runs this and hits
               StackOverflowError
8          System.out.println(out);     // never reached in buggy run
9      }
10
11     static void permute(char[] a, int i, List<String> out) {
12         if (i == a.length) {
13             out.add(new String(a));
```

```
14              return;
15          }
16          for (int j = i; j < a.length; j++) {
17              swap(a, i, j);
18              permute(a, i, out);      // BUG: should be i+1 -> infinite
                    recursion at same i
19              swap(a, i, j);
20          }
21      }
22
23      static void swap(char[] a, int i, int j) {
24          char t = a[i]; a[i] = a[j]; a[j] = t;
25      }
26  }
```

Listing 4.2: Example of buggy permutations (causes stack overflow)

5 Architecture

This chapter describes the architecture of the capture and replay system in JavaWiz, focusing on its core components and their interactions. It outlines how the system captures, stores, and replays Java program executions, emphasizing the design choices made to ensure accurate debugging.

5.1 Overview

The architecture addresses two main challenges in deterministic replay, namely capturing and storing nondeterministic program behavior, and managing file system state. To handle these challenges, we introduce a `Trace System` component (illustrated in Figure 5.1) that manages the capture and replay mechanism. The system is composed of the following subcomponents:

- **Trace Log**: This component stores an ordered sequence of log events that represent nondeterministic operations observed during program execution. During replay, the Trace Log provides these values in the original order to guarantee deterministic results.
- **Virtual File Environment**: This component is an in-memory file system that holds the file state during recording and replay of the executed program. It allows file operations to be captured, restored, and replayed without relying on the actual file system of the host machine.
- **Tracer**: This component is the central coordination component that controls the lifecycle of a trace session. It interacts with both the `Trace Log` and the `Virtual File Environment` to keep the trace data and the state of the file system consistent.

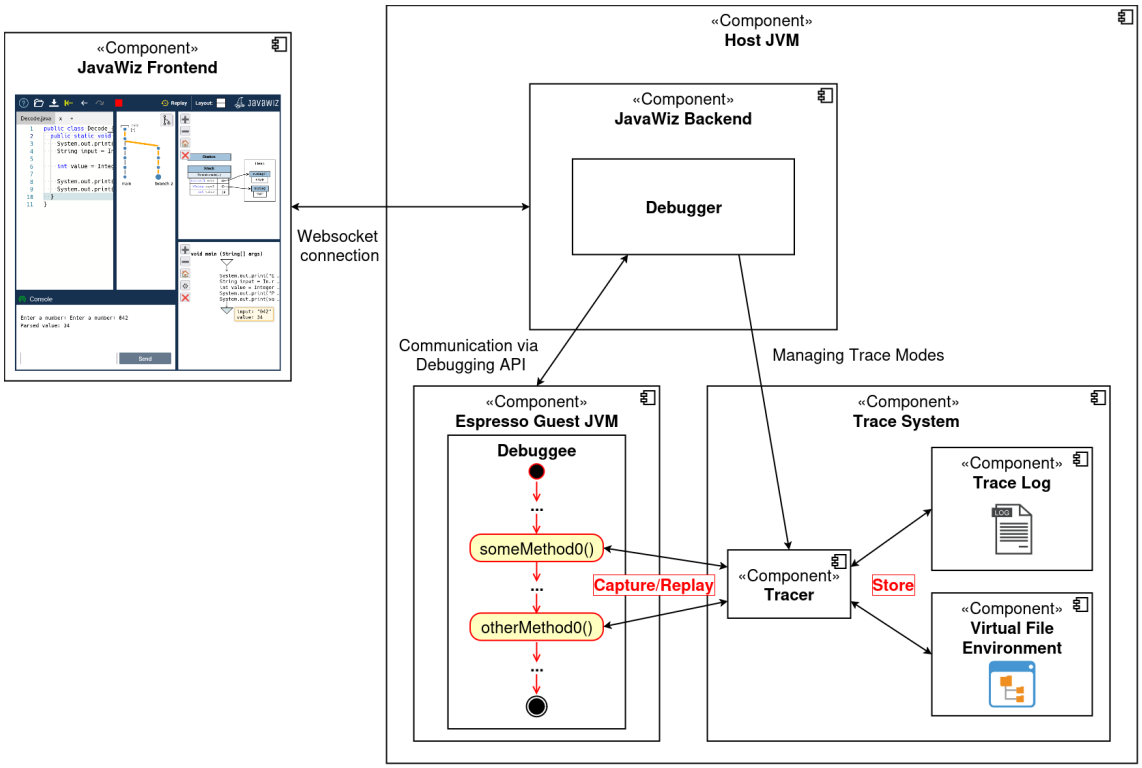


Figure 5.1: Architecture diagram showing interaction of components.

Together, these components form a controlled execution environment in which every recorded input, file change, and nondeterministic effect can be restored with full fidelity, regardless of changes in the host environment.

5.2 Trace Log

The `Trace Log` is the event log of the system. It records every nondeterministic event in the order in which it was observed, together with metadata that allows these events to be unambiguously replayed. During a replay, the log guarantees that the same sequence of values is returned, instead of re-executing the underlying operations. Thus, making program execution deterministic even when the underlying operations are inherently variable. This guarantees that calls to nondeterministic operations (e.g. `System.currentTimeMillis()`, `In.readLine()`, etc.) return exactly the same results as in the original execution.

An essential feature of the Trace Log is its awareness of branching. Instead of maintaining only a single sequence of events, it can hold multiple histories, one for each execution branch. This enables the replay system to capture alternative runs in parallel, without overwriting or discarding the original data. The Trace Log can be serialized and persisted on disk and restored later, ensuring that complete execution sessions can be reconstructed at any time from any machine in JavaWiz.

5.3 Virtual File Environment

The `Virtual File Environment` provides an in-memory file system in which all file operations of the program are captured and can be replayed. By redirecting reads and writes from the host file system, it isolates execution from external influences and prevents inconsistencies caused by changes in the environment. From the perspective of the program, the virtual store behaves like a fully functional file system that supports the creation, deletion, and modification of files and directories.

This component also supports persistence. At the end of a recording session, the file store can be written to disk and later restored during replay. This ensures that the program always encounters the exact same file system state, regardless of what may have changed in the host environment.

5.4 Tracer

The `Tracer` acts as the control unit of our trace system. Its primary responsibility is to decide whether the system is currently capturing new events or replaying previously recorded ones, and to maintain the overall consistency of the execution environment. Rather than simply logging events, it ensures that every nondeterministic outcome is either stored for later use or replaced with its recorded counterpart during replay.

Beyond mode control, the `Tracer` also coordinates branching. As presented in Section 4.1 and 4.2, users can fork the execution into a new branch and continue with alternative inputs or code paths. The Tracer guarantees that each branch progresses deterministically by maintaining a separate event history, while still allowing the user to return to the original

path. In this way, it provides the foundation for exploratory debugging, where multiple alternative executions can be inspected side by side without restarting the program.

By mediating between program execution, the `Trace Log` and the `Virtual File Environment`, the `Tracer` ensure that nondeterministic effects are always captured or reproduced accordingly, keeping the replay environment faithful to the original run.

6 Implementation

This chapter presents the implementation of the capture and replay system in JavaWiz. It describes how the core components are implemented in JavaWiz and how they are integrated into GraalVM Espresso to provide deterministic replay. The chapter then outlines the execution workflows for recording and replaying program runs, discusses advanced capabilities such as branching, and concludes with collaboration and sharing aspects.

6.1 Core Components

The capture and replay system is built around the three main components, the `Tracer`, the `Trace Log` and the `Virtual File Environment` introduced in Chapter 5. Figure 6.1 shows the implemented classes and their relationships. The `Tracer` serves as the central coordinator, interacting with both the `TraceBuffer` (which manages the recorded `TraceEntry` instances) and the `InMemoryFileSystem`, which encapsulates the virtual file environment. The in-memory file system itself is composed of several collaborating classes, including the `InMemoryFileSystemProvider`, `InMemoryFileStore`, `InMemoryPath`, and `InMemorySeekableByteChannel`.

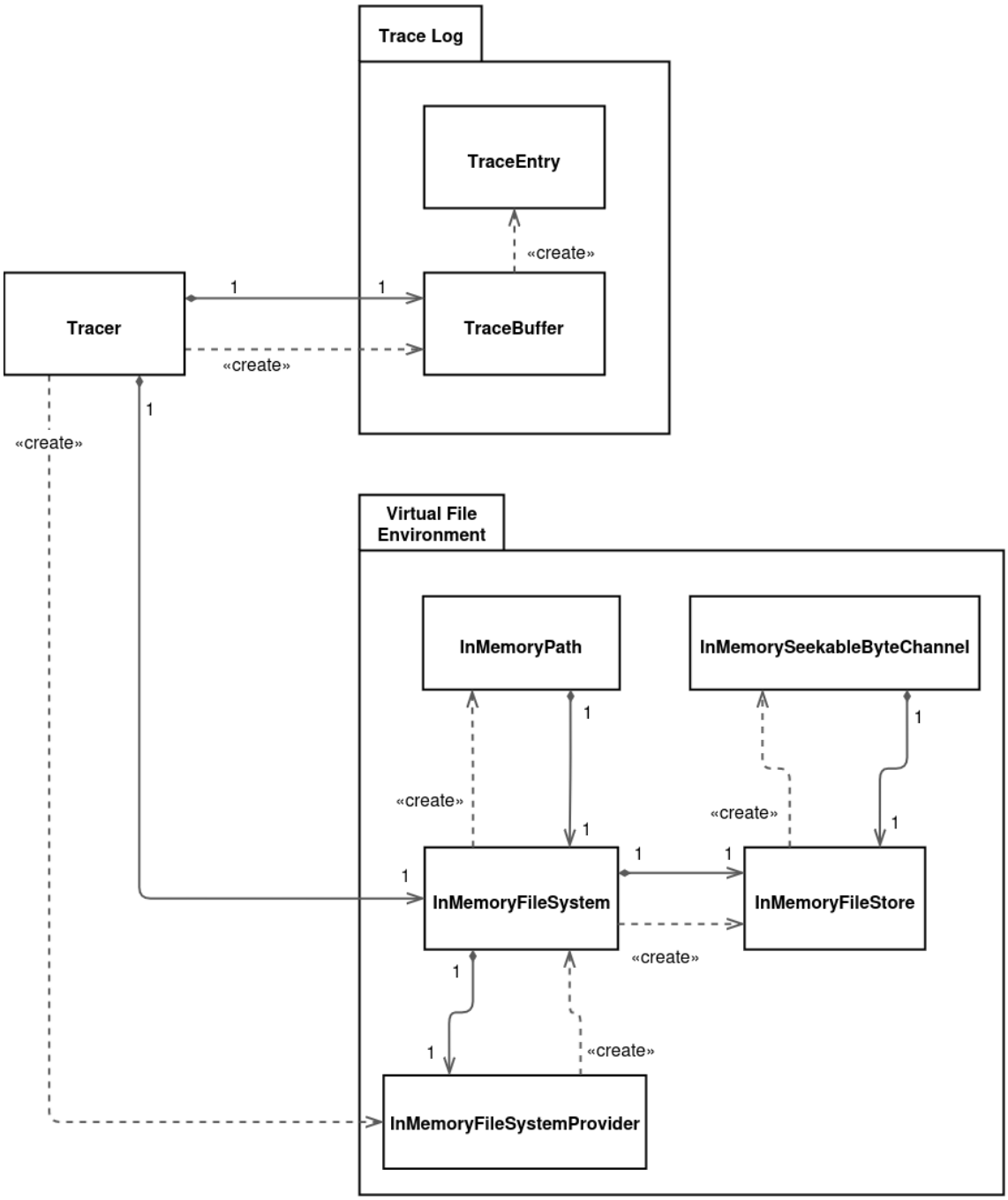


Figure 6.1: Class diagram of the core components of the JavaWiz Trace System.

6.1.1 Trace Log Implementation

The `Trace Log` is realized by two classes:

- The `TraceEntry`, which represents a single recorded value.
- The `TraceBuffer`, which manages ordered collections of these entries.

Trace Entry Data Structure

A `TraceEntry` (Listing 6.1) encapsulates a single recorded value along with its type information. It implements the `Serializable` interface to allow persistence using the Java serialization mechanism. Each entry stores the actual value and its class type, enabling type-safe retrieval during replay.

```
public class TraceEntry implements Serializable {
    private Serializable value;
    private Class<?> type;

    public TraceEntry(Serializable value) {
        this.value = value;
        this.type = value.getClass();
    }

    public Serializable getValue() { return value; }
    public Class<?> getType() { return type; }
}
```

Listing 6.1: TraceEntry data structure.

Branch-Aware Trace Entry Management

The `TraceBuffer` (Listing 6.2) organizes trace entries in branch-specific queues. Each branch is identified by a string key that maps to a `Queue` of entries (first-in-first-out). This allows deterministic consumption of events in the same order in which they were recorded. Persistence is implemented using the Java serialization API [21], allowing traces to be written to files and reloaded for replay (Listing 6.3).

```
public class TraceBuffer implements Serializable {
    private Map<String, Queue<TraceEntry>> buffer = new HashMap<>();
    ...
    private Queue<TraceEntry> get(String branch) {
        return buffer.computeIfAbsent(branch, k -> new LinkedList<>());
    }

    protected void record(String branch, Serializable value) {
        get(branch).add(new TraceEntry(value));
    }

    protected TraceEntry getNextValue(String branch) {
        Queue<TraceEntry> traceQueue = buffer.get(branch);
        if (traceQueue != null && !traceQueue.isEmpty()) {
            return traceQueue.remove();
        }
        return null;
    }
    ...
}
```

Listing 6.2: Branch aware record and consume of values.

```
public class TraceBuffer implements Serializable {
    ...
    protected void persistToFile(Path path) throws IOException {
        try (ObjectOutputStream out = new ObjectOutputStream(new
            FileOutputStream(path.toString()))) {
            out.writeObject(this.buffer);
        }
        ...
    }

    protected void loadFromFile(Path path)
            throws IOException, ClassNotFoundException {
        try (ObjectInputStream in = new ObjectInputStream(new
            FileInputStream(path.toString()))) {
            this.buffer =
                (Map<String, Queue<TraceEntry>>) in.readObject();
        }
        ...
    }
    ...
```

```
}
```

Listing 6.3: Persist and reload of the Buffer.

This implementation ensures that nondeterministic values are captured once during recording and can be retrieved deterministically during replay. By associating entries with execution branches, the buffer also supports branching execution paths, where multiple alternative replays can evolve independently from the same recording root.

6.1.2 Virtual File Environment Implementation

The Virtual File Environment implements a complete Java NIO-compatible file system [22] backed entirely in memory. It consists of a `FileSystemProvider`, a `FileSystem` wrapper around a serializable `FileStore`, a custom `Path` type, and a growable `SeekableByteChannel`. All file operations are routed through these components, allowing the tracer to persist, restore, and replay the file state deterministically.

Provider and channel routing

The entry point in the `InMemoryFileSystemProvider`, which exposes the custom URI scheme `memory://`. Its role is to instantiate the in-memory file system and to route `newByteChannel(...)` calls to the underlying store. Listing 6.4 shows the initialization logic and the scheme registration:

```
public class InMemoryFileSystemProvider extends FileSystemProvider {
    ...
    @Override
    public String getScheme() { return "memory"; }

    @Override
    public FileSystem newFileSystem(URI uri, Map<String,?> env) {
        if (fileSystems.containsKey(uri)) {
            throw new FileSystemAlreadyExistsException();
        }
        if(!getScheme().equals(uri.getScheme())) {
            throw new IllegalArgumentException(
                "Unsupported scheme: " + uri.getScheme());
        }
```

```
        FileSystem fs = new InMemoryFileSystem(this, uri);
        fileSystems.put(uri, fs);
        return fs;
    }
    ...
}
```

Listing 6.4: Registering the `memory://` scheme.

The provider also centralizes the handling of the options for file channels. As shown in Listing 6.5, it interprets NIO open options such as `CREATE`, `APPEND`, or `TRUNCATE`, and dispatches to the appropriate store method.

```
public class InMemoryFileSystemProvider extends FileSystemProvider {
    ...
    @Override
    public SeekableByteChannel newByteChannel(Path path,
            Set<? extends OpenOption> options, FileAttribute<?>... attrs)
            throws IOException {
        InMemoryPath p = (InMemoryPath) path;
        InMemoryFileSystem fs = (InMemoryFileSystem) p.getFileSystem();
        InMemoryFileStore store = (InMemoryFileStore) fs.getFileStore();

        if (options.contains(CREATE_NEW)) {
            return store.createFile(p.toString());
        } else if (options.contains(CREATE) || options.contains(WRITE)) {
            if (options.contains(TRUNCATE_EXISTING)) {
                return store.truncateFile(p.toString());
            }
            return store.openFile(p.toString(), options.contains(APPEND));
        } else if (options.contains(READ) || options.isEmpty()) {
            return store.openFile(p.toString(), false);
        }
        throw new UnsupportedOperationException("Unsupported: " + options);
    }
    ...
}
```

Listing 6.5: Routing open options to the store.

Therefore, all file reads and writes follow a single controlled code path, which is crucial for the deterministic replay mechanism.

Serializable store

The `InMemoryFileStore` holds file contents in a concurrent map. Each key is a file path, and the value is a byte array representing its contents. Listing 6.6 shows the data structure and file creation method.

Opening, truncating or updating files works similarly by replacing or retrieving entries in the map. Moreover, the store can be serialized and deserialized to a file using Java serialization similar to the `TraceBuffer`, as shown in Listing 6.7. This allows the complete file system state to be persisted at the end of a recording and be reloaded during replay.

```
public class InMemoryFileStore extends FileStore implements Serializable {
    ...
    private ConcurrentHashMap<String, byte[]> files =
        new ConcurrentHashMap<>();

    public SeekableByteChannel createFile(String path) throws IOException {
        if (files.containsKey(path)) {
            throw new FileAlreadyExistsException(path);
        }
        byte[] bytes = new byte[0];
        files.put(path, bytes);
        return new InMemorySeekableByteChannel(bytes, this, path, false);
    }
    ...
}
```

Listing 6.6: File store backed by a concurrent map.

```
public class InMemoryFileStore extends FileStore implements Serializable {
    ...
    public void saveToFile(Path path) throws IOException {
        try (var oos = new ObjectOutputStream(
                new FileOutputStream(path.toString()))) {
            oos.writeObject(this);
        }
    }

    public void loadFromFile(Path path) {
        try (var ois = new ObjectInputStream(
                new FileInputStream(path.toString()))) {
            InMemoryFileStore store = (InMemoryFileStore) ois.readObject();
```

```
            this.files = store.files;
        }
        ...
    }
    ...
}
```

Listing 6.7: Persist and reload the file store.

Seekable, growable byte channel

Every open file is represented by an `InMemorySeekableByteChannel`. The ByteChannel implements the Java NIO interfaces (`read`, `write`, `truncate`) to guarantee that the in-memory representation always reflects the file state. Listing 6.8 shows the grow-on-write behavior: the backing array expands as needed, and the store is updated after every mutation.

```
public class InMemorySeekableByteChannel implements SeekableByteChannel {
    ...
    private byte[] data;
    private int position;
    private boolean open;
    private final InMemoryFileStore fileStore;
    private final String path;

    public InMemorySeekableByteChannel(byte[] data,
        InMemoryFileStore fileStore, String path, boolean append) {
            this.data = data;
            this.position = append ? data.length : 0;
            this.open = true;
            this.fileStore = fileStore;
            this.path = path;
    }
    ...
    @Override
    public int write(ByteBuffer src) throws IOException {
        ensureOpen();
        int n = src.remaining();
        ensureCapacity(position + n);
        src.get(data, position, n);
```

```
24          position += n;
25          store.updateFile(path, data);
26          return n;
27      }
28
29      private void ensureCapacity(int required) {
30          if (required > data.length)
31              data = Arrays.copyOf(data, required);
32      }
33      ...
34  }
```

Listing 6.8: Grow-on-write semantics.

Truncate operations (Listing 6.9) shrink the backing array and adjust the position accordingly.

```
1   public class InMemorySeekableByteChannel implements SeekableByteChannel {
2       ...
3       @Override
4       public SeekableByteChannel truncate(long size) throws IOException {
5           ensureOpen();
6           if (size < data.length)
7               data = Arrays.copyOf(data, (int) size);
8           if (position > size) position = (int) size;
9           store.updateFile(path, data);
10          return this;
11      }
12      ...
13  }
```

Listing 6.9: Truncating a file.

File system wrapper

The `InMemoryFileSystem` ties provider and store together and acts as an API for the tracer to reset the file system between sessions, or to persist and restore the virtual file data. Listing 6.10 shows the relevant methods:

```
1   public class InMemoryFileSystem extends FileSystem {
2       ...
```

```
    public void clearFileStore() { fileStore.clearStore(); }
    public void initStoreFromFile(Path path)
            throws IOException, ClassNotFoundException {
        fileStore.loadFromFile(path);
    }
    public void persistStoreToFile(Path path)
            throws IOException {
        fileStore.saveToFile(path);
    }
    ...
}
```

Listing 6.10: API for managing the underlying file store.

Minimal path representation

Finally, `InMemoryPath` provides just enough functionality for the provider and the byte channel. Its `toUri()` implementation (Listing 6.11) ensures that the file paths are bound to the `memory://` scheme.

```
public class InMemoryPath implements Path {
    ...
    private final InMemoryFileSystem fileSystem;
    private final String path;

    @Override
    public URI toUri() {
        try {
            return new URI(fs.provider().getScheme(), null, path, null);
        } catch (URISyntaxException e) {
            throw new RuntimeException("Invalid URI: " + path, e);
        }
    }
    ...
}
```

Listing 6.11: Scheme-aware path URIs.

6.1.3 Tracer Implementation

The `Tracer` is the central coordinator of our tracing system. It coordinates the `Trace Log`, the `Virtual File Environment`, and the guest program executed in Espresso. Its responsibility is to ensure that every nondeterministic operation is either captured or reproduced, depending on the current execution mode.

The tracer operates in three modes: `OFF`, `RECORD`, and `REPLAY`, represented by an internal `TraceMode` enum (Listing 6.12). Mode switching is controlled by the JavaWiz Backend and is handled by lifecycle methods such as `startRecording()`, `continueReplaying()`, and `initReplaySession()`.

```
public class Tracer {
    ...
    static enum TraceMode {
        OFF,
        RECORD,
        REPLAY;
    }

    private static final TraceBuffer recordBuffer = new TraceBuffer();
    private static final TraceBuffer replayBuffer = new TraceBuffer();

    public static void initRecordingSession() {
        recordBuffer.clear();
        replayBuffer.clear();
        inMemoryFileSystem.clearFileStore();
        setTraceMode(Tracer.TraceMode.RECORD);
    }

    public static void initReplaySession(Path path) {
        initRecordingSession();
        recordBuffer.loadFromFile(path.resolve("trace_log.bin"));
        inMemoryFileSystem.initStoreFromFile(path.resolve("trace_filestore.
            bin"));
        continueReplaying();
    }

    public static void continueReplaying() {
        setTraceMode(TraceMode.REPLAY);
        copyTraceBuffer(recordBuffer, replayBuffer);
```

```
29      }
30      ...
31  }
```

Listing 6.12: Tracer execution mode handling

In record mode, methods that return nondeterministic values are intercepted (explained later in Section 6.2) and logged. In replay mode, the tracer retrieves the previously captured values and substitutes them in place of the original calls. This behavior is encapsulated by the `trace()` and `reproduce()` methods (Listing 6.13), which form the primary API for all replay-aware substitutions.

```
1  public class Tracer {
2      ...
3      private static String currentBranch;
4
5      public static void trace(String clazz, String function, Serializable
           value) {
6          recordBuffer.record(currentBranch, clazz, function, value);
7      }
8      public static <T> T reproduce(String function) {
9          TraceEntry entry = replayBuffer.getNextValue(currentBranch,
               function);
10         return (T) entry.getType().cast(entry.getValue());
11     }
12     public static void setCurrentBranch(String branch) {
13         Tracer.currentBranch = branch;
14     }
15     ...
16 }
```

Listing 6.13: Tracer recording values when in record mode and reproducing them when replaying.

Record and replay buffers

The tracer maintains two separate trace buffers: a *record buffer* and a *replay buffer* (Listing 6.12). The record buffer collects events during execution, while the replay buffer is a working copy created at the beginning of each replay session. This distinction is necessary because, during replay sessions, events are consumed from the replay buffer in the exact

order they were originally recorded. By isolating consumption in a dedicated replay buffer, the record buffer remains intact as the central log.

Figure 6.2 illustrates the interaction between JavaWiz, the tracer, the trace buffers and the debuggee, during a replay session. The process begins when the JavaWiz backend invokes `initReplaySession()` on the tracer. At this point, the tracer resets its state, loads the recording into the record buffer, and copies the contents of the record buffer into the replay buffer (Listing 6.14). The execution mode is then set to *REPLAY*.

When the replay is executed, each nondeterministic operation is intercepted and routed through the tracer. For every intercepted call, the tracer queries the replay buffer with `getNextValue()`. If a corresponding `TraceEntry` exists, its value is retrieved and returned to the debuggee. This process continues until the replay buffer is empty or bypassed.

The following scenarios can exhaust or bypass the replay buffer:

1. If the program reaches the end during recording, the replay simply finishes as well.
2. If the recording ended in the middle of an execution, the replay buffer eventually runs out of events while the program is still running. At this point, the tracer transparently switches back to record mode, capturing the remainder of the execution into the record buffer.
3. If the user deliberately branches into previously unexplored code during replay, the buffer may not yet be exhausted, but from the point of divergence onward, the system must switch back to recording. New events are appended to the record buffer while preserving the original trace.

In any case, the record buffer remains intact as the main log, ensuring that the replay can always resume from the original recording. Without two buffers, consumed entries would be lost and branching scenarios would not be possible.

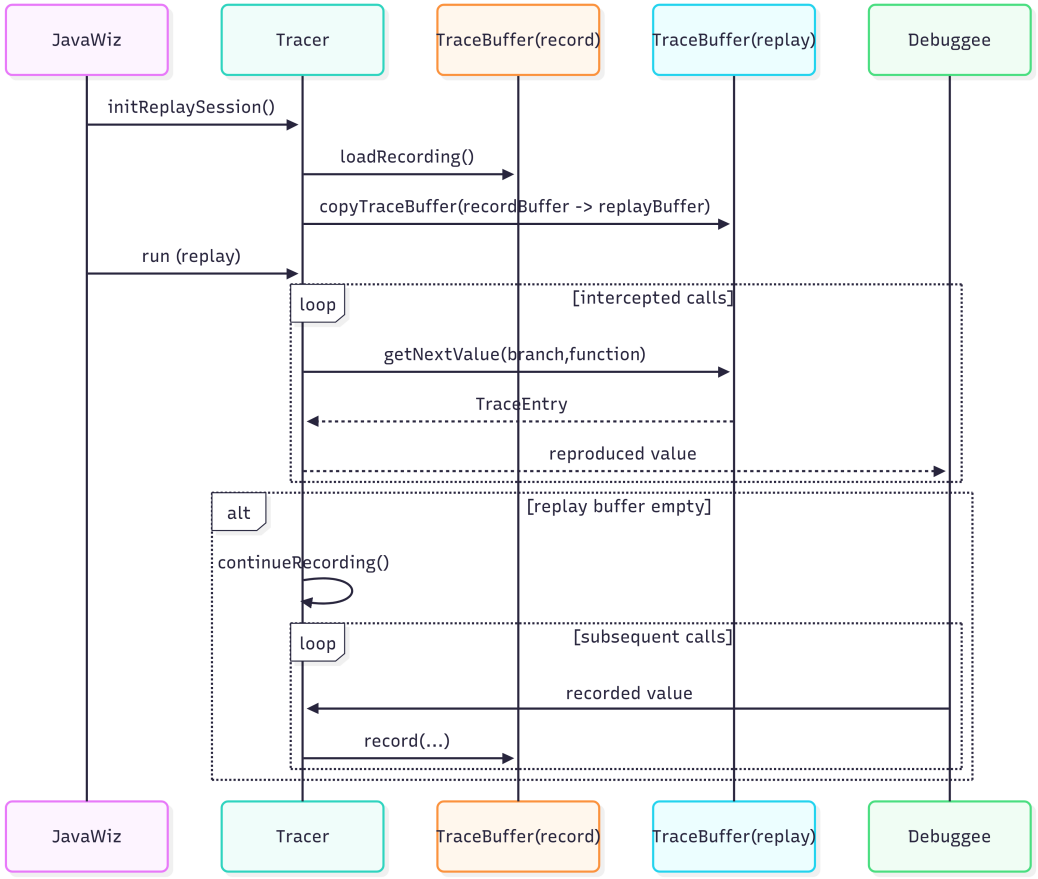


Figure 6.2: Replay with separate buffers: values are consumed from the replay buffer while the record buffer remains intact as the canonical source.

```
1 public class Tracer {
2     ...
3     private static void copyTraceBuffer(TraceBuffer source, TraceBuffer
          destination) {
4         for (Map.Entry<String, Queue<TraceEntry>> entry : source.getBuffer
              ().entrySet()) {
5             Queue<TraceEntry> newQueue = new LinkedList<>(entry.getValue())
                  ;
6             destination.getBuffer().put(entry.getKey(), newQueue);
7         }
8     }
9     ...
```

```
10 }
```

Listing 6.14: Copying the record buffer into a dedicated replay buffer.

Branching support

Branching allows the user to fork execution at any program state during debugging. Each branch is identified by a unique branch name and newly recorded events are tagged accordingly in the trace buffer (as described in Section 6.1.1). Events that occurred before the branch remain associated with the original branch, while subsequent operations evolve along the newly created path.

Switching branches during replay is coordinated by JavaWiz. For each trace state retrieved during replay, if the branch of the current and previous `traceState` differs, the JavaWiz backend invokes `Tracer.setCurrentBranch(traceState.branchname)` (Listing 6.13), instructing the tracer to consume or record events from the chosen branch.

Selective tracing with Truffle

When recording a running program, we need to avoid tracing events that are invoked by the Java standard library or the runtime itself, as this would quickly pollute the trace with irrelevant events and introduce unnecessary overhead. Such executions are not part of the JavaWiz debuggee code, for example, class loading by the bootstrap class loader [23] during the JVM startup may trigger file I/O operations, which are also nondeterministic. To address this, the `shouldTraceNode()` method ensures that only the debuggee code is traced (Listing 6.15). By traversing the stack frames with `Truffle.getRuntime().iterateFrames()`, we employ a `FrameInstanceVisitor` (the functional interface passed into iterateFrames) [24] to walk over the current execution frames in Espresso. Through this visitor, the tracer inspects the active call node and checks whether it belongs to the method being traced, which is the `main()`-method of the JavaWiz debuggee. Tracing is then enabled exclusively for the truffle nodes within the debuggee code.

```
public class Tracer {
    ...
    private static String tracedNode;
    // traceNode: "main([Ljava/lang/String;)V"
    ...
    public static boolean shouldTraceNode() {
        return Boolean.TRUE.equals(
            Truffle.getRuntime().iterateFrames(frameInstance -> {
                Node callNode = frameInstance.getCallNode();
                if (callNode != null) {
                    RootNode rootNode = callNode.getRootNode();
                    if (rootNode.getName().equals(tracedNode)) {
                        return true; // found frame
                    }
                }
                return null; // continue iteration
            })
        );
    }
    ...
}
```

Listing 6.15: Selective tracing of debuggee code only.

Persistence management

Finally, the tracer also manages persistence. The recorded traces are always written to fixed file names (`trace.bin` for the trace log, `trace_filestore.bin` for the file store). This choice simplifies reproducibility by initializing replay sessions directly from a dedicated recording file (see Section 6.4) containing these files. The tracer provides a `save()` method to persist both components and an `initReplaySession()` method to restore them (Listing 6.12).

6.2 Intercepting with `@Substitution`

A crucial part of the implementation is the ability to intercept method calls, particularly native methods calls, inside Espresso and redirect them through the `Tracer`. This is made possible by the `Substitution` mechanism provided by Espresso [7, 17].

The `@Substitution` annotation allows standard library methods to be effectively replaced with code that is run on the host machine. Using this option, we can modify and substitute methods in the JDK with capture-and-replay-aware versions. These substituted methods delegate their behavior to the `Tracer` acting as interceptors, ensuring that nondeterministic results are captured during recording and faithfully reproduced during replay.

Example and Usage

In the standard JDK, `System.currentTimeMillis()` is declared as a native method, as shown in Listing 6.16.

```
// java/lang/System.java
public final class System {
...
    public static native long currentTimeMillis();
...
}
```

Listing 6.16: Declaration of System.currentTimeMillis() in the standard JDK.

On HotSpot, this declaration is linked to a C/C++ implementation of the method through JNI. When called, the JVM dispatches into the operating system to obtain the current time.

Espresso, in contrast, avoids JNI overhead for selected methods by substituting them with direct calls to the corresponding host JVM implementations (see Listing 6.17), increasing performance in Espresso. However, Espresso is still capable of invoking native code via JNI when necessary.

```
import com.oracle.truffle.espresso.substitutions.*;
import com.oracle.truffle.api.CompilerDirectives;

@EspressoSubstitutions
```

```
final class Target_java_lang_System {

    @Substitution(hasReceiver = false)
    public static long currentTimeMillis() {
        // Replace the native call with system call of host JVM
        return java.lang.System.currentTimeMillis();
    }
}
```

Listing 6.17: Example: Substituting System.currentTimeMillis().

Listing 6.18 shows a guest program invoking `System.currentTimeMillis()` exactly as it would on a HotSpot JVM. From the perspective of the guest program, the substitution is completely transparent: the call to `System.currentTimeMillis()` appears as if it were resolved through the standard native interface without changes of the semantics of the call.

```
public class TimeProbe {
    public static void main(String[] args) throws InterruptedException {
        long t0 = System.currentTimeMillis();
        Thread.sleep(25);
        long t1 = System.currentTimeMillis();

        System.out.println("delta(ms)=" + (t1 - t0));
        System.out.println("now(ms)=" + t1);
    }
}
```

Listing 6.18: Guest code that executes the substituted native method.

Tracing Integration in Espresso

To enable deterministic recording and replay, the substituted target methods must integrate the `Tracer`, by wrapping logic that checks the current execution mode. Each substitution follows a structured pattern.

- **Replay:** If the replay is active and the current node belongs to the guest program under trace, the substitution fetches the next recorded value from the replay buffer instead of executing the operation again.

- **Record:** If recording is active, the substitution executes the actual operation on the host JVM, captures its result, and stores it in the trace log, tagged with the method signature and branch.
- **Bypass:** If tracing is disabled, the method executes normally without recording or replaying.

The flowchart in Figure 6.3 illustrates this tracing pattern. For each target method, the tracer first checks whether replay mode is enabled, if so, the substitution immediately returns the recorded result. If not, the actual operation is executed. In record mode, the produced value from the actual operation is captured by the tracer before being returned as in a normal execution. This decision flow guarantees that nondeterministic operations behave consistently across replay sessions.

In Listing 6.19 the tracer-aware implementation of `System.currentTimeMillis()` in Espresso is demonstated:

```
@EspressoSubstitutions
final class Target_java_lang_System {
    public static long currentTimeMillis() {
        if (Tracer.isReplay() && Tracer.hasRemainingTrace() && Tracer.
            shouldTraceNode()) {
            return Tracer.reproduce("currentTimeMillis");
        }
        long millis = System.currentTimeMillis();
        if (Tracer.isRecord() && Tracer.shouldTraceNode()) {
            Tracer.trace(System.class.getName(), "currentTimeMillis",
                millis);
        }
        return millis;
    }
}
```

Listing 6.19: Tracer-aware substitution of System.currentTimeMillis().

When in replay mode, the substitution retrieves the next recorded value that matches the function name from the replay buffer. If in record mode, it executes the host call, captures the result, and appends it to the trace log. The `shouldTraceNode()` guard ensures that only the guest program code (normally the `main()` method of the debuggee) is traced, avoiding pollution of the log with internal JDK or runtime calls.

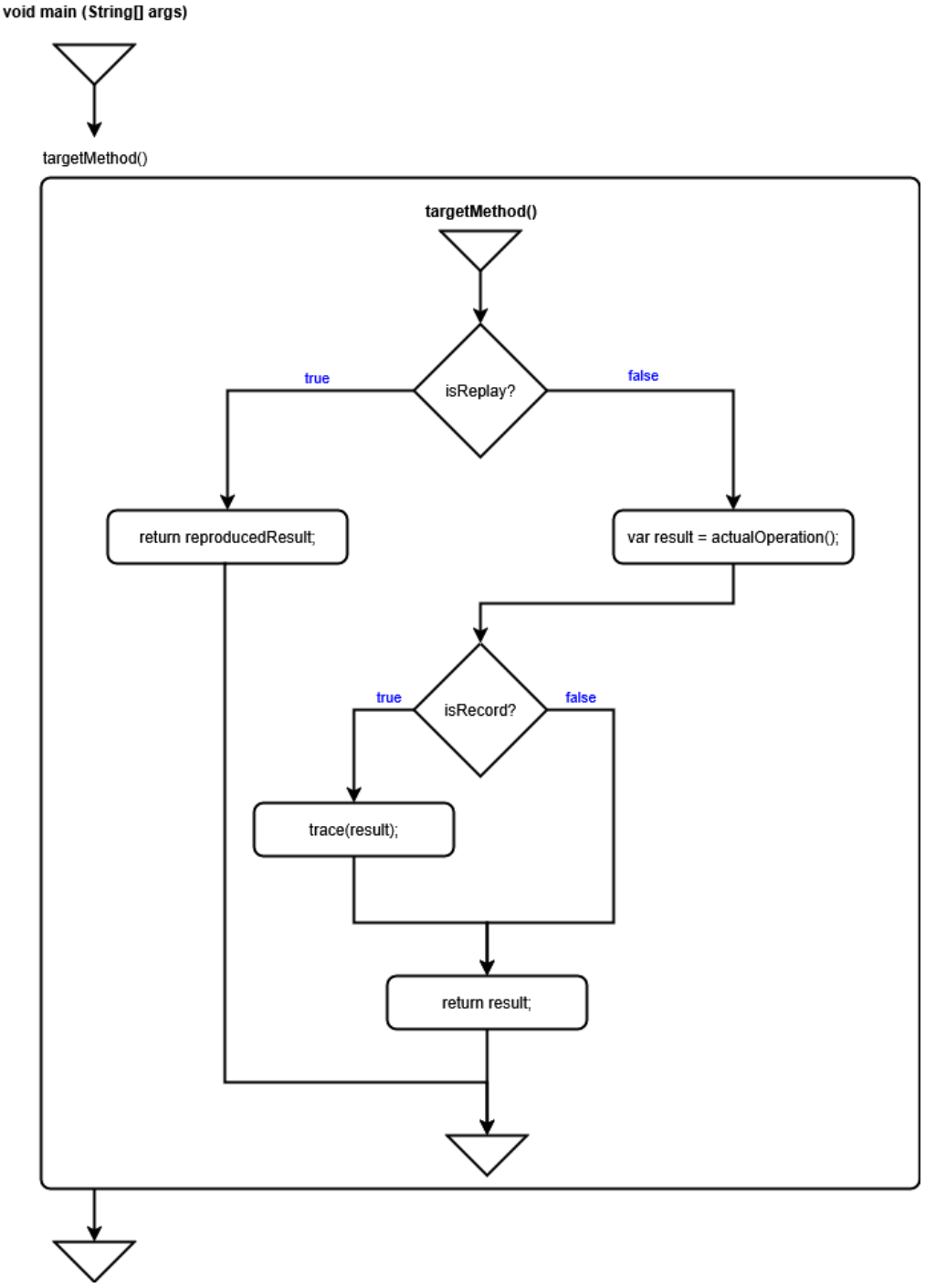


Figure 6.3: Tracing flow: replayed values are returned if available, otherwise the real operation is executed and optionally traced.

This integration guarantees that repeated executions of the same guest program will return the exact same timestamps during replay, even if the underlying host system time has advanced. The same substitution pattern is applied to other nondeterministic calls, such as user inputs, environment variable access, or file I/O operations.

6.3 Step Back to Previous States

A key improvement of our approach is the introduction of a `live step back` functionality. In earlier versions of JavaWiz, stepping back was limited to a static visualization. Users could browse past states in the execution trace, but the underlying program execution was not actually restored. In contrast, the current system restores both the program state and the file system state, enabling execution to continue seamlessly from an earlier point as if no interruption had occurred.

In JavaWiz, this is realized by restarting the debuggee program from the beginning and re-executing it deterministically in the Espresso backend, until the desired program state is reached. Since all nondeterministic operations (e.g. input handling) are replayed using values from the trace log, the re-execution exactly reproduces the original behavior. By following this approach, correctness is maintained while remaining compatible with our trace-based architecture of JavaWiz is guarantied.

Figure 6.4 illustrates the workflow for stepping back:

(1) The user selects a previous program state in the execution trace tree (state `s6`) via a click in the frontend.

(2) The backend restarts the debuggee in Espresso and initializes the replay mode by copying the `RecordBuffer` into the `ReplayBuffer` and restoring the initial file state in the virtual file system.

(3) The program is deterministically re-executed, consuming values from the `ReplayBuffer` (reproduced input: `"Java"`) has been processed until the selected target state is reached. Once the target state is reproduced, the user can resume execution. This may either continue replay along the original path (reproduce input: `"Wiz"`) or branch to input and record new events into an alternative execution path.

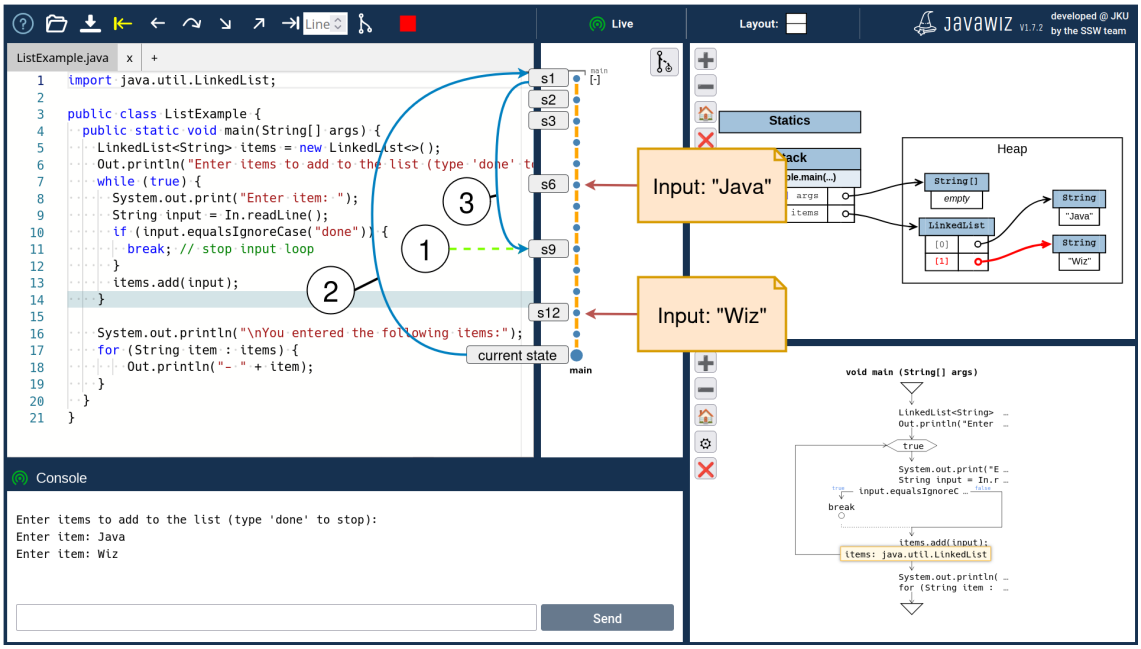


Figure 6.4: Workflow of the `step back` functionality in JavaWiz. The user selects an earlier state, the system deterministically replays execution up to that point, and execution continues from there.

Restarting the debuggee in the Espresso backend, we can send a step-back request directly via the Web Worker API.

Figure 6.5 illustrates the backend sequence of operations:

1. The frontend issues the step-back request.
2. The backend creates a response builder that checks if the target state has been reached.
3. The backend stores the information of the previous traces (undo trace) from the request.
4. The backend restarts the Espresso debuggee.
5. The backend executes the debuggee deterministically.
6. The response builder returns a result once the target state is reached.
7. The frontend updates the user interface to match the target state.

8. The backend blocks itself by setting the `SharedArrayBuffer` [5] and waits for the next request.

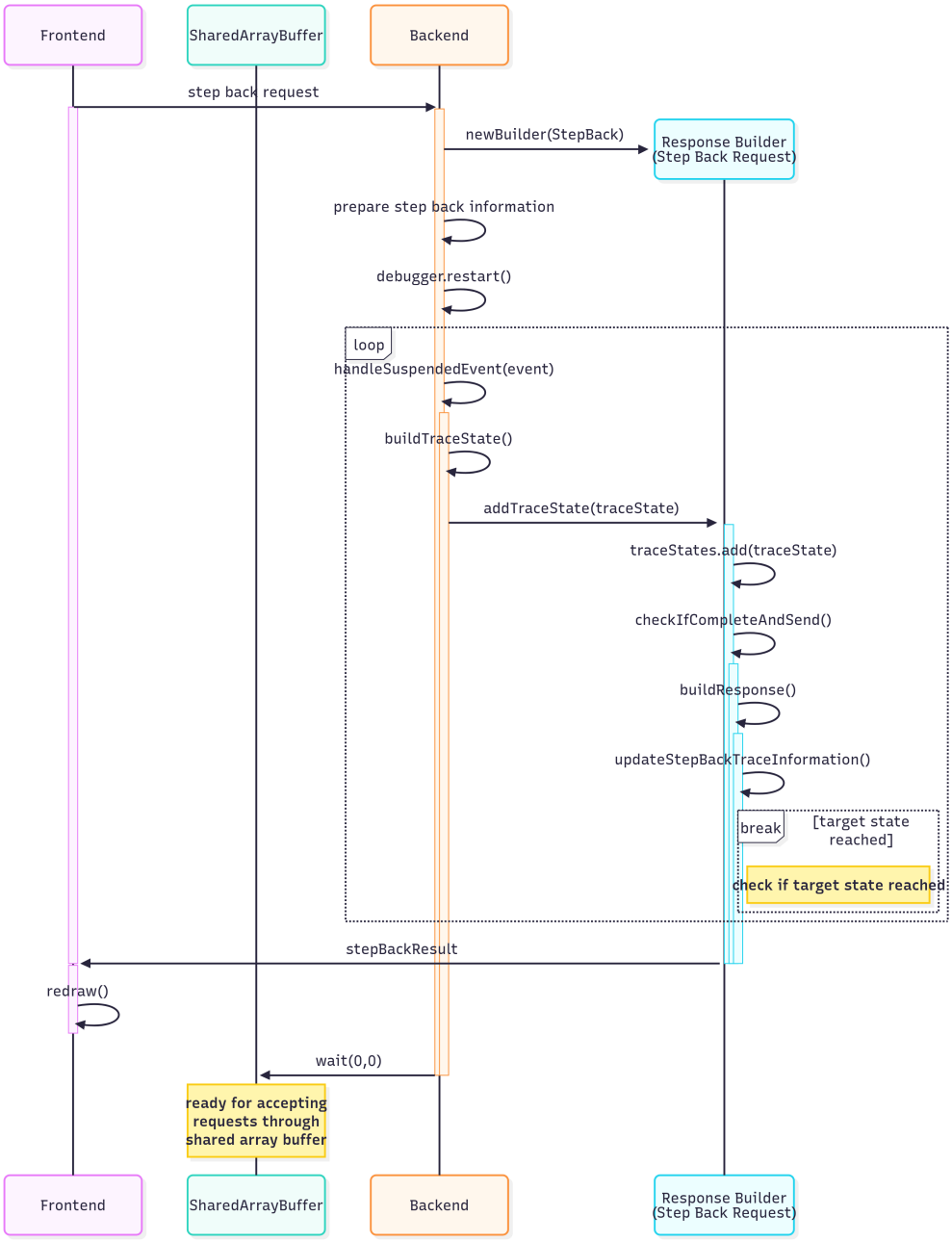


Figure 6.5: Sequence diagram of the step-back execution.

Listing 6.20 shows the entry point for handling a step-back request in the backend. When a `StepBackRequest` is received, the backend sets up the undo trace information and restarts the debugger to perform a deterministic re-execution.

```
fun processRequest(request: Request): Response? {
    when (request) {
        ...
        is StepBackRequest -> {
            latestDebugger?.let { debugger ->
                if (compiledWithWebImage() &&
                    debugger.vm is EspressoVirtualMachine) {
                    EspressoTraceHelper
                        .withStepBackInformation(request, debugger.vm)
                }
                debugger.restart() // deterministic re-run of debugger
                ...
            }
        }
        ...
    }
}
```

Listing 6.20: Handling a step-back request by setting stepBackInformation (undo trace) and restarting.

During replay, a helper utility tracks the current step-back steps with a thread-local holder (Listing 6.21). After each state transition, the helper advances toward the target state.

```
object EspressoTraceHelper {
    private val espressoTraceStateInformation = ThreadLocal<
        EspressoTraceStateInformation?>()
    ...
    fun withStepBackInformation(request: StepBackRequest,
                                vm: EspressoVirtualMachine) {
        val stepBackTask = request.toDebuggerStepBackTask()
        vm.skipTracing = true
        val espressoStepBackTraceStateInformation =
            EspressoTraceStateInformation(vm, stepBackTask)
        espressoTraceStateInformation
            .set(espressoStepBackTraceStateInformation)
    }
    fun updateStepBackTraceInformation() {
        val info = espressoTraceStateInformation.get()
```

```
        if (info?.debuggerTask is DebuggerStepBackTask) {
            info.nextTraceState()
        }
    }
    ...
}
```

Listing 6.21: Advancing step-back metadata during replay.

The target condition for completing a step back is defined at the task level. The step-back task specifies that the operation is complete once the number of replayed steps is equal to the size of the undo trace (Listing 6.22).

```
data class StepBackTask(
    override val latestUndoTrace: MutableList<PreviousTrace>,
    override val branch: String
) : DebuggerStepBackTask(latestUndoTrace, branch) {
    override val task: DebuggerTaskKind = DebuggerTaskKind.STEP_BACK
    override fun targetReached(stepCount: Int): Boolean =
        latestUndoTrace.size == stepCount
}
```

Listing 6.22: StepBackTask: target condition for completing a step-back

The Espresso debugger encapsulates in `EspressoTraceStateInformation`, how to move along the undo trace, and also instructs Espresso which branch to *checkout* ensuring the execution is along the correct branch (Listing 6.23).

```
data class EspressoTraceStateInformation(
    val vm: EspressoVirtualMachine,
    val debuggerTask: DebuggerTask,
    private var currentStateIdx: Int = 0
) {
    fun nextTraceState() {
        val prevTrace = debuggerTask.latestUndoTrace[currentStateIdx]
        ...
        if (debuggerTask is DebuggerStepBackTask) {
            if (currentStateIdx < debuggerTask.latestUndoTrace.size - 1) {
                currentStateIdx++
                vm.setCurrentBranch(prevTrace.branchName)  // select branch
            } else {
                vm.setCurrentBranch(debuggerTask.branch)
                vm.skipTracing = false
```

```
16              }
17          }
18      }
19  }
```

Listing 6.23: VM-directed navigation to earlier states.

Because all nondeterministic events are already captured in the trace, it is not necessary to reconstruct each intermediate trace state. Intermediate states would be indistinguishable from their previously recorded counterparts and recomputing them would only add overhead.

To address this, the backend creates dummy trace states during step-back operations. These placeholders represent replayed execution without building the full trace state (Listing 6.24). By doing so, the system can advance efficiently through replay while preserving a consistent trace index for the target condition.

```
1  fun buildTraceState(event: SuspendedEvent,
2                      esprRunner: EspressoRunner): TraceState? {
3      ...
4      if (esprRunner.vm.skipTracing) {
5          return TraceState.dummy()
6      }
7      ...
8  }
```

Listing 6.24: Dummy trace states are created.

During this process, the backend continuously checks whether the selected target state has been reached. Once the target condition is met, the step-back operation is complete and a `StepBackResultResponse` is sent to the frontend (Listing 6.25).

```
1  fun buildResponse(): Response? {
2      when (request) {
3          ...
4          is StepBackRequest -> {
5              updateStepBackTraceInformation()
6              if (!isVMRunning ||
7                  isWaitingForInput ||
8                  (latestStepBackTask != null && latestStepBackTask!!.
                       targetReached(traceStates.count()))
9              ) {
```

```
10                  return StepBackResultResponse(
11                      request, StepResult(emptyList(), isWaitingForInput,
                            isVMRunning)
12                  )
13              }
14          }
15          ...
16      }
17  }
```

Listing 6.25: Condition and completion for step back.

6.4 Sharing and Collaboration

The implementation of sharing in JavaWiz is realized by packaging all trace-related artifacts into a single transferable file and exposing it as a URL. A recording consists of the following key artifacts:

- The `trace.bin` containing the trace log.
- The `trace_filestore.bin` capturing the state of the virtual file system.
- The `trace_tree.json` that encodes the execution trace tree (shown in Figure 4.8) as a JSON representation.
- The `user source files`.

When the user requests a sharing of the execution, the frontend receives a `ShareResponse` with the `trace.bin` and the `trace_filestore.bin` produced by the backend. The frontend then:

1. Creates a ZIP containing the trace artifacts.
2. Generates a UUID-based filename and either uploads the ZIP to the server.
3. Shows a share URL: `<Domain of JavaWiz>/share/<UUID>` (shown in Figure 4.9).

When a user opens a shared URL, the archive is fetched and unpacked. The `tracer` is initialized with the restored `trace.bin` and `trace_filestore.bin` in the backend. The `trace_tree.json` is reloaded into session storage to restore the execution trace tree, to show the recorded states visually. Finally, all source files are restored into the workspace editor. After these steps, the session is immediately replayable, all nondeterministic effects are served from the `tracer` reproducing the original run.

With this implementation, collaboration is simplified to a reproducible workflow. One user records and shares an execution, while another retrieves and replays it deterministically. The trace log guarantees that nondeterministic values are reproduced exactly, and the in-memory file store ensures an identical file system state. This combination allows students or lecturers to explore, step through, and branch from the same execution state without requiring access to the original environment. Figure 6.6 illustrates this workflow.

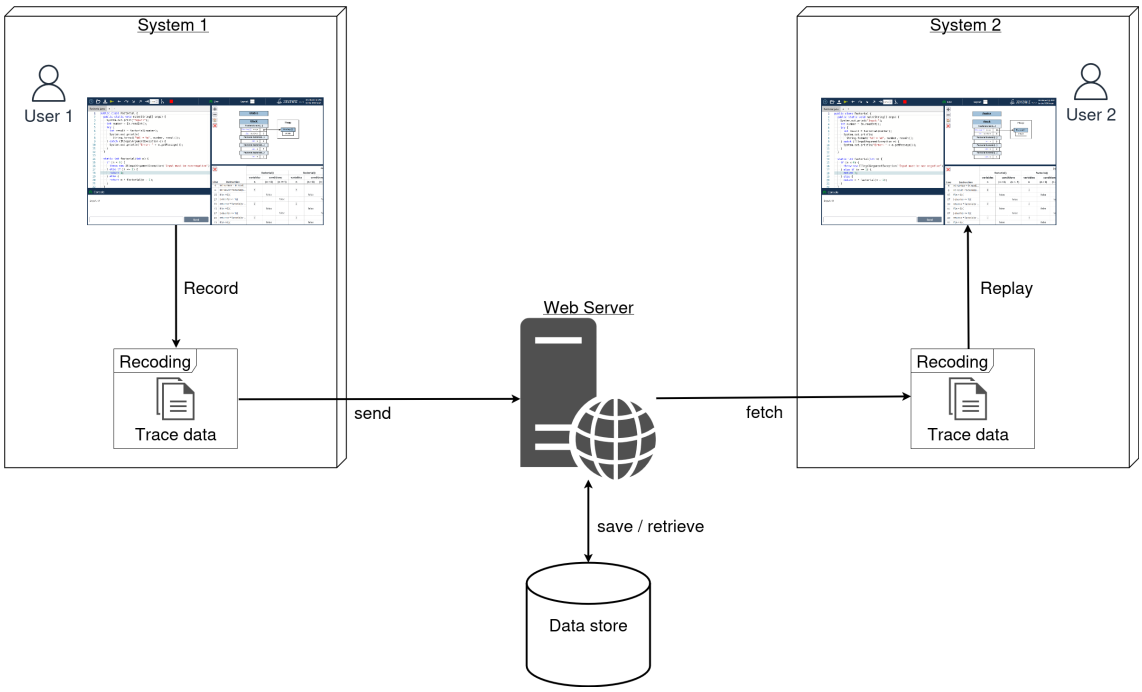


Figure 6.6: Sharing workflow. User 1 records an execution and uploads the recording to the web server. User 2 fetches the recording and replays the execution.

7 Discussion

This chapter reflects on the implementation decisions made in our presented capture and replay system and analyzes their implications. We highlight the limitations of the current implementation and discuss a possible complementary technique using the GraalVM Continuation API [25] to archive *true state snapshots*. Together, these perspectives outline where the system can be improved and where directions for further research are.

7.1 Limitations

The deterministic replay model in JavaWiz is effective for debugging sequential programs whose nondeterminism arises from system calls such as system time, user input, or file I/O. The architecture and implementation demonstrate that deterministic replay of single-threaded Java programs is both feasible and useful in teaching scenarios. However, when our approach is extended to more complex programs, several challenges arise.

This section outlines two more advanced areas in software development that are also relevant in educational contexts: multithreading and concurrency and network I/O. Both introduce nondeterminism that the current system cannot fully control, and therefore challenge the assumptions of the presented approach.

7.1.1 Multithreading and Concurrency

Multithreaded programs introduce nondeterminism not only through calls of native methods but also through the way threads interleave their execution in the JVM. To achieve deterministic replay in such cases, the trace system would need to capture the

precise scheduling decisions that occurred during recording and enforce the same order during replay.

Our capture and replay architecture is based on sequential execution of the program. It records nondeterministic values such as system time, file I/O, and user input, but does not trace or enforce the order of thread execution. As a result, a replay that faithfully reproduces the recorded values may still diverge once multiple threads run concurrently.

The first limitation lies in ordering. The `Trace Log` maintains one FIFO queue per branch during recording. However, it does not include a thread identification mechanism or an explicit record of the execution schedule. If two or more threads reach substituted calls in a different order during replay, the next value consumed from the queue may be attributed to the wrong thread. The two-buffer arrangement (described in Section 6.1.3) assumes a single consumer. When multiple threads try to consume recorded values simultaneously, the notion of reproducing the next event becomes ambiguous without a recorded schedule.

The second limitation is synchronization. Deterministic concurrency requires capturing *happens-before relationships* [26] induced by operations such as `Thread.start/join`, monitor enter/exit (`synchronized`, `wait/notify`), `Lock/Condition`, volatile reads and writes, atomics and memory fences. None of these effects are currently intercepted or timestamped, and occur inside JDK libraries that JavaWiz deliberately excludes from tracing. Consequently, multiple code paths that resolved one way during recording may resolve differently during replay, even with identical trace values.

Consequently, the virtual file environment poses additional concurrency challenges. While the in-memory store itself uses concurrent data structures, open channels expose mutable byte arrays without external coordination. Concurrent writes to the same file path can interleave in ways that neither captures nor replays, since file-system atomicity and locking semantics are not modeled in conjunction with thread scheduling.

Overall, in the context of education, our current design is well-suited for single-threaded programs and teaching scenarios where reproducible I/O, time, and input are the focus. Applied to multithreaded code, it should be regarded as best-effort replay that is useful for illustrating local effects but not guaranteed to reproduce schedules faithfully. Full determinism would require per-thread logs, recorded synchronization events, logical clocks, and scheduling capable of enforcing the captured order.

7.1.2 Networking I/O

Input and output operations, such as reading from or writing to network sockets, or databases, can introduce nondeterministic behavior. The system would need to log these operations and their results to ensure consistent replay.

Although our tracing mechanism works well for the deterministic replay of local nondeterministic operations such as file I/O, time queries, or user input, it is not an ideal approach for capturing network I/O. Network interactions introduce several challenges that limit the effectiveness of method-level interception.

The fundamental issue is that network communication depends on external servers or services, which may be unavailable, produce large or continuous data streams, or evolve their state independently of the debuggee. This makes deterministic replay inherently problematic. Furthermore, branching into unexplored code paths that involve network calls cannot be resolved deterministically, since the recorded trace contains no results for those alternative requests.

The following examples illustrate where `@Substitution` is not an ideal approach:

Server Availability

One limitation arises when the remote server that was contacted during the original execution is no longer available at replay time. In such cases, even though the initial request and its response can be captured and replayed deterministically, any new requests introduced cannot be served live since the server cannot be reached.

Listing 7.1 shows a simplified example of this problem. A request is made to `https://api.javawiz.net/data` using the Java HTTP client. During the original run, this request may succeed and its response body can be captured by the tracer for later replay. However, if the server is offline or permanently unavailable during a subsequent replay, the call to `httpClient.send(...)` will throw an `IOException`.

```
1 HttpRequest req = HttpRequest.newBuilder()
2     .uri(URI.create("https://api.javawiz.net/data"))
3     .build();
4
```

```
try {
    HttpResponse<String> response = httpClient.send(req, BodyHandlers.
        ofString());
    // response can be reproduced
} catch (IOException e) {
    System.err.println("Server not reachable: " + e.getMessage());
    // subsequent live requests could result in exceptions
}
```

Listing 7.1: HTTP request failing when the server is no longer available

In this example, the replay system can reproduce the originally captured `response` by substituting the native call, returning values from the trace log. But if the user continues execution past the recorded events or deliberately branches into a new path that requires another request to the same server, the replay fails because the live server is not available anymore.

Unlike deterministic file I/O in the `In-Memory File Store`, where the complete state can be captured and restored consistently, the correctness of network I/O depends on the availability and behavior of external servers, which can evolve independently or cease to exist. This makes deterministic replay fundamentally limited in such scenarios.

Database Connections

Database drivers (e.g. JDBC) communicate with remote servers over TCP. Substituting query results introduce limitations in the logical assumptions of the execution path. Even if queries and results are captured, replay cannot guarantee consistency with the live database state once branching or step-back is introduced. The fundamental problem is that, unlike the in-memory file store, the database state is not captured as part of the trace.

Consider the following sequence of SQL statements executed against a live database shown in Listing 7.2.

```
-- main branch: Statement 1
INSERT INTO items (id, name) VALUES (1, 'foo');

-- main branch: Statement 2
SELECT * FROM items WHERE name = 'foo';  -- returns 'foo'
```

```
6
7 -- main branch: Statement 3
8 DELETE FROM items WHERE name = 'foo';
```

Listing 7.2: Sequential SQL statements in the main execution

Suppose that the user steps back to *main branch: Statement 2* during replay (Listing 7.3). At this point, the original replay would substitute the recorded result, returning `'foo'` as stored in the trace. If the user then creates a branch and continues execution from that point, divergence occurs.

```
1 -- main branch: Statement 2 (replay)
2 SELECT * FROM items WHERE name = 'foo';  -- substituted: returns 'foo'
3
4 -- branch: Statement 3 (executed live)
5 DELETE FROM items WHERE id = 1;     -- live DB read, 'foo' no longer exists
```

Listing 7.3: Branch replay after stepping back

Here, the replayed branch assumes that the value `'foo'` exists (because the substitution injected it), but the live database no longer contains it since the original Statement 3 already deleted it. The branch and the real database state are inconsistent, and deterministic replay cannot be maintained.

This stands in sharp contrast to the `Virtual File Environment` (Section 5.3). In that case, the initial file system state is captured during recording and restored at the beginning of a replay session. When stepping back or creating a new branch, the file system can be rewound or duplicated consistently, ensuring that all paths evolve deterministically from the same initial state in educational scenarios. However, for databases, such capturing would require capturing the full contents and transactional state of the database engine, which is typically infeasible.

As a result, direct substitution for database I/O is insufficient. Without capturing and restoring the full database state (for example, by introducing database rollback mechanisms), step-back and branching scenarios inevitably break determinism.

Authentication and Session Tokens

Authentication flows introduce one of the most difficult challenges for deterministic replay. Many services issue ephemeral session identifiers, cookies or JSON Web Tokens (JWT) that expire quickly or are valid only once. Although our tracing system can capture and replay the token itself, the underlying server state cannot be reproduced deterministically.

During replay, the client can resend the captured token, but the server will typically reject it because it has already expired, has been revoked, or is tied to a session that no longer exists. This problem becomes even more apparent when branching. If the user steps back in the execution and issues new requests, the trace cannot produce new valid tokens without contacting the live server.

```
1  HttpResponse<String> res = httpClient.send(request, BodyHandlers.ofString()
       );
2
3  // Token is ephemeral and only valid for the original session
4  String token = res.headers().firstValue("Authorization").orElseThrow();
5
6  // Store it in trace during recording
7  Tracer.trace(HttpClient.class.getName(), "authToken", token);
8
9  //replay ends here
10 ...
```

Listing 7.4: Capturing an authentication token from an HTTP response

If we suppose a login API (Listing 7.4) that returns a token that is valid for only 15 minutes. In the original execution, subsequent requests succeed because the token is still fresh.

In scenarios where the system replays the captured token and the user continues the execution (either by exhausting the trace log or by branching), the authentication may be accepted if the trace is recent but will fail if the session has already expired on the server.

7.2 Espresso Continuations

Beyond the method-level tracing approach, one could imagine capturing the execution state of a program directly. Continuations in GraalVM offer such a mechanism, enabling

suspension and resumption of execution at specific points. This section explores how continuations could serve as an alternative or complementary model for deterministic replay in JavaWiz.

Conceptually, a continuation packages the current call stack, so that the program can be suspended, persisted, and resumed later. In Espresso, this is implemented by copying guest-side stack frames into host-side objects called HostFrameRecords (HFRs) [25, 27]. These records form a linked list, effectively capturing the program state and creating a "time-travel" snapshot for the program state.

Illustrative example

Listing 7.5 shows the continuation model. The guest program cooperates by calling `suspend()` at safe points, a continuation object is then serialized and restored to resume execution.

```
import org.graalvm.continuations.*;
import java.io.*;
import java.nio.file.*;

public class ContinuationApp implements ContinuationEntryPoint,
    Serializable {

    @Override
    public void start(SuspendCapability suspendCap) {
        System.out.println("Step 1: Start execution");
        suspendCap.suspend(); // snapshot here
        System.out.println("Step 2: Resumed after first suspend");
        suspendCap.suspend(); // snapshot again
        System.out.println("Step 3: Resumed after second suspend");
    }

    public static void main(String[] args) throws Exception {
        Continuation c = loadOrInit();
        c.resume();       // advance to next suspension point (or finish)
        save(c);          // persist continuation for later
    }

    static Continuation loadOrInit() throws Exception {
        Path p = Path.of("continuation_state.bin");
```

```
24          return Files.exists(p) ? load(p) : Continuation.create(new
                ContinuationApp());
25      }
26
27      static Continuation load(Path p) throws IOException,
            ClassNotFoundException {
28          try (var in = new ObjectInputStream(Files.newInputStream(p))) {
29              return (Continuation) in.readObject();
30          }
31      }
32
33      static void save(Continuation c) throws IOException {
34          try (var out = new ObjectOutputStream(Files.newOutputStream(Path.of
                ("continuation_state.bin")))) {
35              out.writeObject(c);
36          }
37      }
38  }
```

Listing 7.5: Snapshotting with continuations (illustrative).

Potential approach and limitations

In the context of capture and replay executions in JavaWiz, continuations promise precise and compact state snapshots at explicit suspension points. Currently, stepping back to a previous point in JavaWiz works by restarting the program and re-executing deterministically with the captured values until the desired state is reached (as described in Section 6.3).

With continuations, the debugger could either load the previous state that included a nondeterministic event and deterministically run to the desired state or simply load the desired execution state directly, without restarting and replaying from the beginning (see Figure 7.1). This could reduce latency and improve the user experience when exploring earlier execution states, compared to our current workflow (described previously in Figure 6.4 in Section 6.3).

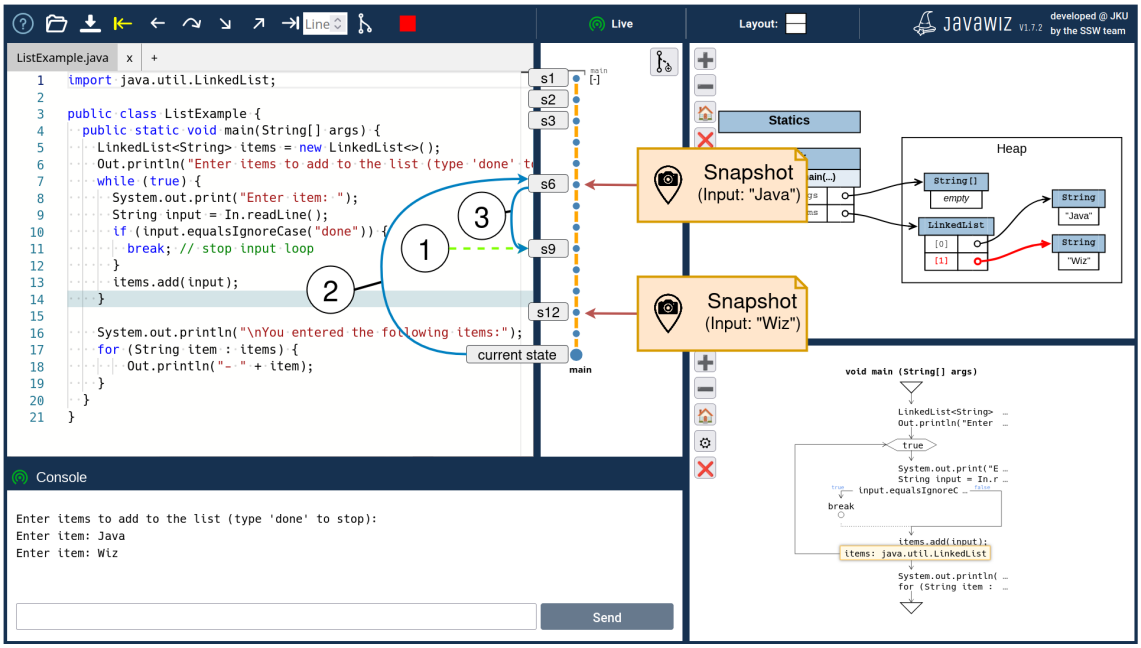


Figure 7.1: Theoretical workflow of the `step back` functionality in JavaWiz realized with the Continuation API. The user selects an earlier state, the system loads the previous snapshot from this state and executes up to the selected state, and execution continues from there.

However, at the time of writing this thesis, practical use is constrained by several factors:

- **Guest code involvement:** Suspension points must be placed in the debuggee (guest) program. That implies instrumenting source code or bytecode (e.g. to inject calls to `suspend()`), which conflicts with our current approach of using substitutions as interceptors Espresso (described in Section 6.2), to replay arbitrary student programs without altering them in JavaWiz. Without modifying guest code or weaving in bytecode, no snapshot can be taken.
- **External resources:** Continuations do not serialize external effects (e.g. file descriptors, OS resources, etc.). A snapshot must be paired with a model for these effects (e.g. separate snapshot of the virtual file environment).
- **Security:** Continuations are safe while kept inside the VM. Security concerns arise when continuations are `serialized` (materialized) and later `deserialized` (dematerialized) [25, 27]. Since the Espresso JVM performs only minimal validation during deserialization, an attacker can craft a continuation, injecting malicious bytecode. Consequently, continuations shall not be loaded from untrusted sources (e.g. shared JavaWiz recordings).

8 Related Work

This chapter reviews existing approaches and tools related to the capture and replay functionality in debugging. It discusses the state of the art in debugging technologies, highlighting their strengths and limitations, and positions the proposed enhancements within this landscape.

8.1 DRD of SoftPLC Programs

Prähofer et al. [18, 28] present a deterministic replay debugging technique tailored for diagnosing bugs in deployed SoftPLC (software-based programmable logic controller) applications, particularly those developed using IEC 61131-3 languages and operating under strict real-time constraints. Their method captures all external inputs and sources of nondeterminism during runtime, enabling an exact replay of the program execution in an offline development environment. This facilitates in-depth debugging without requiring access to the original hardware setup. The approach emphasizes minimal performance overhead by leveraging the structured nature of IEC 61131-3 programs, ensuring real-time execution remains unaffected. Rather than logging detailed scheduling information, the system reconstructs task interleaving based on data dependencies, streamlining the logging process. Periodic snapshots of the internal state of the system further support flexible and efficient replay. The authors provide an automated toolchain, including a time-traveling debugger, and validate their solution through industrial case studies, demonstrating its practicality and effectiveness for real-world SoftPLC debugging scenarios.

8.2 Omniscient Debugging

Bil Lewis [29] introduces in his paper the concept of omniscient debugging, a method that records every state change and method call in a program, allowing developers to navigate "backwards in time" to inspect any past state without rerunning the program. This approach addresses the limitations of traditional breakpoint debuggers by removing guesswork about where to set breakpoints, avoiding missed states, and also enabling deterministic debugging. The paper also gives insight into the implementation, the Omniscient Debugger (ODB), that is written in Java and works by instrumenting bytecode to log all variable assignments and method calls, each tagged with a timestamp. Its graphical interface supports precise navigation through a history of the program, object state inspection, and selective display of changed variables. Lewis categorizes bugs into "seen snake in the grass" cases, where faulty output can be traced backward to its cause. And "unseen snake" cases, where no direct faulty output is present. User studies showed that ODB users could locate the source of certain bugs in minutes compared to hours with conventional tools. Although memory constraints in a 32-bit environment limit recording to about 10 million events, strategies such as selective instrumentation, garbage collection of old events, and triggered recording make the approach practical for larger programs back then. Performance tests indicate acceptable overhead for many real-world scenarios, and the design is extensible to larger address spaces. Compared to related work in reversible execution and deterministic replay, the uniqueness of ODB lies in its state-oriented presentation, time-based navigation, and integration with query-based event analysis.

8.3 BugVis

BugVis [30, 31] is a debugger designed specifically for students and novice programmers, aiming to simplify the often complex debugging process through intuitive and customizable visualizations. Developed in response to common frustrations with traditional debuggers, BugVis introduces blackboard-style representations of data structures that closely resemble classroom diagrams, making it easier for learners to grasp program behavior. A standout feature is its support for time-travel debugging, which allows users to step backward and forward through the execution of a program to better understand

how bugs emerge and evolve. The tool prioritizes ease of use, reducing dependence on print statements by offering clear, educationally grounded visual feedback. Although still under development, BugVis is built on solid design principles intended to bridge the gap between programming education and practical debugging, presenting a promising advancement for beginner-friendly development tools.

8.4 Undo

Undo [32] is a product suite specializes in time-travel debugging. Their core offering, UDB (Undo Debugger), is an interactive time-traveling debugger for C, C++, and Rust. It allows developers to record live program execution or attach to running processes, then replay the execution forwards and backwards to inspect program states and trace from symptom to root cause with GDB (GNU Debugger) [33] compatibility and IDE integrations such as Visual Studio Code and Jetbrains' CLion. For Java and Kotlin, LiveRecorder [34] offers similar capabilities, developers can record execution (e.g. in Continuous Integration (CI) or automated tests), then replay with reverse stepping controls (step-into, step-out, step-over) to fully examine and debug failures like concurrency exceptions, data-structure corruption, or logic errors. Across all products, Undo emphasizes on reducing debug cycles, making intermittent or hard-to-reproduce bugs reproducible and diagnosable within a single record-replay cycle, essentially allowing developers to "wind back the clock" through program execution history.

8.5 IntelliTrace

IntelliTrace [35, 36, 37] is a debugging tool integrated into Visual Studio, designed to enhance application lifecycle management by enabling developers to record and replay code execution events, which helps in diagnosing and resolving complex bugs more efficiently. Introduced in Visual Studio 2010, IntelliTrace allows users to step backward in time during a debugging session, capturing detailed historical information such as method calls, exceptions, and variable values, which traditional debuggers often miss. This feature streamlines the debugging process, reduces the time spent reproducing issues, and improves collaboration among development teams by providing a clear record of application

behavior. IntelliTrace is particularly valuable in large, complex projects where intermittent or hard-to-reproduce bugs can significantly delay development. Its integration with other Visual Studio tools, such as unit testing, code analysis, and performance profiling, further supports a comprehensive approach to software quality and lifecycle management. The tool is highlighted as a key innovation in the Visual Studio suite of Application Lifecycle Management (ALM) features, making advanced debugging accessible to a broader range of developers and teams.

9 Conclusion and Future Work

This thesis presented the design and implementation of a capture and replay mechanism for JavaWiz with the aim of improving programming education. The work addressed the limitations of conventional debugging tools by introducing deterministic replay debugging into JavaWiz, leveraging the GraalVM Espresso interpreter and the Truffle framework. In addition, our system also adapted the integration with GraalVM Web Image, the JavaScript interoperability API, enabling full execution and visualization in the browser.

The core contribution of this research lies in extending JavaWiz with a system that records nondeterministic events, such as user input, system calls, and time-dependent operations, and accurately replays them to reproduce program executions. The developed architecture, consisting of the Tracer, Trace Log, and Virtual File Environment, ensures that recorded executions can be replayed with full fidelity, enabling reproducibility, collaborative debugging, and effective teaching scenarios. Additionally, the integration of branching and sharing functionality empowers students and instructors to explore alternative program paths and exchange execution traces, enhancing both exploration and communication in learning contexts.

Through this work, JavaWiz has evolved from a visualization tool to a comprehensive platform that not only illustrates program behavior but also enables reproducible and shareable execution analysis. The system demonstrates strong potential to support programming education by providing instructors with deterministic teaching material and students with interactive opportunities to understand code behavior and debug effectively.

However, certain limitations remain. The handling of concurrency of multithreaded programs or networking I/O poses challenges for ensuring full determinism, and future work could extend the replay model to capture these aspects. Furthermore, utilizing

the GraalVM Continuation API presents an opportunity to implement state snapshots, allowing users to resume execution from arbitrary points in time with greater flexibility.

In summary, this thesis contributes to a foundation for reproducible debugging in JavaWiz, enriching its role as an educational tool. By combining deterministic replay with intuitive visualization, it offers a meaningful step toward bridging the gap between abstract runtime concepts and practical understanding in programming education.

Bibliography

[1] Katrin Anna Kern. *JavaWiz – A Visualization Tool for Software Development Education*. eng. 2023. URL: https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-65592 (cit. on pp. 1, 5, 6).

[2] Markus Weninger, Simon Grünbacher, and Herbert Prähofer. "JavaWiz: A Trace-Based Graphical Debugger for Software Development Education". In: *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. 2025, pp. 01–12. DOI: 10.1109/ICPC66645.2025.00023 (cit. on pp. 1, 5).

[3] *Vue.js — vuejs.org*. https://vuejs.org/guide/introduction.html#what-is-vue. [Accessed 18-07-2025] (cit. on p. 1).

[4] *What is D3? | D3 by Observable — d3js.org*. https://d3js.org/what-is-d3. [Accessed 18-07-2025] (cit. on p. 1).

[5] Felix Schenk. *Development of a Java Debugger Framework Based on the Espresso VM and Its Compilation to JavaScript*. eng. 2024. URL: https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-77068 (cit. on pp. 1, 6, 62).

[6] *Java Debug Interface — docs.oracle.com*. https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/. [Accessed 04-08-2025] (cit. on p. 6).

[7] *Espresso — graalvm.org*. https://www.graalvm.org/latest/reference-manual/espresso/. [Accessed 04-08-2025] (cit. on pp. 6, 11, 56).

[8] *Introduction — graalvm.org*. https://www.graalvm.org/latest/introduction/. [Accessed 04-08-2025] (cit. on p. 9).

[9] Matthias Grimmer et al. "High-performance cross-language interoperability in a multi-language runtime". In: *Proceedings of the 11th Symposium on Dynamic Languages*. DLS 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 78–90. ISBN: 9781450336901. DOI: 10.1145/2816707.2816714. URL: https://doi.org/10.1145/2816707.2816714 (cit. on p. 9).

[10] Thomas Würthinger et al. "Self-optimizing AST interpreters". In: *SIGPLAN Not.* 48.2 (Oct. 2012), pp. 73–82. ISSN: 0362-1340. DOI: 10.1145/2480360.2384587. URL: https://doi.org/10.1145/2480360.2384587 (cit. on pp. 9, 10).

[11] Christoph Pichler et al. "Hybrid Execution: Combining Ahead-of-Time and Just-in-Time Compilation". In: *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2023. Cascais, Portugal: Association for Computing Machinery, 2023, pp. 39–49. ISBN: 9798400704017. DOI: 10.1145/3623507.3623554. URL: https://doi.org/10.1145/3623507.3623554 (cit. on p. 9).

[12] Matija Sipek, Branko Mihaljevic, and Aleksander Radovan. *Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM*. Dec. 2021. DOI: 10.48550/arXiv.2112.14716 (cit. on p. 10).

[13] *Truffle Language Implementation Framework — graalvm.org*. https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/. [Accessed 04-08-2025] (cit. on pp. 10, 11, 16).

[14] Christian Wimmer and Thomas Würthinger. "Truffle: a self-optimizing runtime system". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH '12. Tucson, Arizona, USA: Association for Computing Machinery, 2012, pp. 13–14. ISBN: 9781450315630. DOI: 10.1145/2384716.2384723. URL: https://doi.org/10.1145/2384716.2384723 (cit. on p. 10).

[15] Oleg Šelajev. *Java on Truffle - Going Fully Metacircular — medium.com*. https://medium.com/graalvm/java-on-truffle-going-fully-metacircular-215531e3f840. [Accessed 04-08-2025] (cit. on p. 12).

[16] *graal/espresso/README.md at master · oracle/graal — github.com*. https://github.com/oracle/graal/blob/master/espresso/README.md. [Accessed 04-08-2025] (cit. on p. 11).

[17] *graal/espresso/docs/how-espresso-works.md at master · oracle/graal — github.com*. https://github.com/oracle/graal/blob/master/espresso/docs/how-espresso-works.md. [Accessed 04-08-2025] (cit. on pp. 11, 56).

[18] Herbert Prähofer et al. "A Comprehensive Solution for Deterministic Replay Debugging of SoftPLC Applications". In: *IEEE Transactions on Industrial Informatics* 7.4 (2011), pp. 641–651. DOI: 10.1109/TII.2011.2166768 (cit. on pp. 13, 78).

[19] Xu Vyacheslav et al. "ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay". In: () (cit. on p. 13).

[20] *Java Native Interface Specification: 1 - Introduction — docs.oracle.com*. https://docs.oracle.com/en/java/javase/24/docs/specs/jni/intro.html. [Accessed 22-07-2025] (cit. on p. 15).

[21] *Java Object Serialization Specification: 1 - System Architecture — docs.oracle.com*. https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serial-arch.html. [Accessed 21-08-2025] (cit. on p. 42).

[22] *java.nio.file (Java Platform SE 8) — docs.oracle.com*. https://docs.oracle.com/javase/8/docs/api/java/nio/file/package-summary.html. [Accessed 21-08-2025] (cit. on p. 44).

[23] *Chapter 5. Loading, Linking, and Initializing — docs.oracle.com*. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html. [Accessed 08-09-2025] (cit. on p. 54).

[24] *TruffleRuntime (GraalVM Truffle Java API Reference) — graalvm.org*. https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/TruffleRuntime.html#iterateFrames(com.oracle.truffle.api.frame.FrameInstanceVisitor). [Accessed 11-09-2025] (cit. on p. 54).

[25] *Continuation API — graalvm.org*. https://www.graalvm.org/latest/reference-manual/espresso/continuations/. [Accessed 15-09-2025] (cit. on pp. 68, 74, 77).

[26] *Memory Consistency Errors (The Java Tutorials > Essential Java Classes > Concurrency) — docs.oracle.com*. https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html. [Accessed 17-09-2025] (cit. on p. 69).

[27] *graal/espresso/docs/continuations.md at master · oracle/graal — github.com*. https://github.com/oracle/graal/blob/master/espresso/docs/continuations.md. [Accessed 15-09-2025] (cit. on pp. 74, 77).

[28] Herbert Prähofer et al. "Deterministic Replay Debugging of IEC 61131-3 SoftPLC programs". In: *2010 8th IEEE International Conference on Industrial Informatics*. 2010, pp. 1110–1117. DOI: 10.1109/INDIN.2010.5549586 (cit. on p. 78).

[29] Bil Lewis. "Debugging Backwards in Time". In: *CoRR* cs.SE/0310016 (2003). URL: http://arxiv.org/abs/cs/0310016 (cit. on p. 79).

[30] Kim Mönch and Mark Minas. "BugVis: A Debugger Supporting Time-Travel and Data Structure Visualization". In: *Proceedings of the 6th European Conference on Software Engineering Education*. ECSEE '25. Association for Computing Machinery, 2025, pp. 195–199. ISBN: 9798400712821. DOI: 10.1145/3723010.3723032. URL: https://doi.org/10.1145/3723010.3723032 (cit. on p. 79).

[31] Kim Mönch. "Time-Travel Debugging with Visualization of Data-Structures Based on Instrumentation". In: *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2022, pp. 1–3. DOI: 10.1109/VL/HCC53370.2022.9833002 (cit. on p. 79).

[32] *Undo for C, C++, Go and Rust Undo.io — docs.undo.io*. https://docs.undo.io/index.html. [Accessed 10-08-2025] (cit. on p. 80).

[33] *GDB: The GNU Project Debugger — sourceware.org*. https://www.sourceware.org/gdb/. [Accessed 10-08-2025] (cit. on p. 80).

[34] *Getting started - Undo.io 8.2.1-dev.g0c930b3c8d77 documentation — docs.undo.io*. https://docs.undo.io/java/GettingStarted.html. [Accessed 18-08-2025] (cit. on p. 80).

[35] Justin Marks. *Intellitrace - debugging applications with IntelliTrace*. Apr. 2010. URL: https://learn.microsoft.com/en-us/archive/msdn-magazine/2010/april/intellitrace-debugging-applications-with-intellitrace (cit. on p. 80).

[36] Mickey Gousset et al. "Professional Application Lifecycle Management with Visual Studio 2010". In: 2010. URL: https://api.semanticscholar.org/CorpusID:86851824 (cit. on p. 80).

[37] Sandeep Chanda and Damien Foggon. "Application Lifecycle Management". In: *Beginning ASP.NET 4.5 Databases*. Berkeley, CA: Apress, 2013, pp. 235–249. ISBN: 978-1-4302-4381-6. DOI: 10.1007/978-1-4302-4381-6_13. URL: https://doi.org/10.1007/978-1-4302-4381-6_13 (cit. on p. 80).