INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Educational Program Visualizations Using Synthetized Execution Information

Rodrigo Manuel Dias Mourato

Master's in Computer Science and Engineering

Supervisor:
PhD André Leal Santos, Assistant Professor,
Iscte - Instituto Universitário de Lisboa

September, 2024

TECNOLOGIAS
E ARQUITETURA

Department of Information Science and Technology (ISTA)

Educational Program Visualizations Using Synthetized Execution Information

Rodrigo Manuel Dias Mourato

Master's In Computer Science and Engineering

Supervisor:
PhD André Leal Santos, Assistant Professor,
Iscte - Instituto Universitário de Lisboa

September, 2024

"Programming allows you to think about thinking, and while debugging you learn learning."

Nicholas Negroponte

Acknowledgement

I would like to express my sincerest gratitude to all those who have contributed to the completion of this master's dissertation. I would be remiss if I did not first acknowledge my teacher, André Santos, whose unwavering guidance, encouragement, and expertise were instrumental in my academic journey. I am indebted to my mentors for imparting their knowledge and expertise, which have greatly benefited me. Furthermore, I would like to express my gratitude to my fellow students who have been actively involved in this project. I am indebted to my family for their unwavering support and belief in my abilities. I would like to express my gratitude to my girlfriend, who has provided invaluable support and encouragement throughout this process, and who has instilled in me a belief in my capacity to learn and improve. I would also like to acknowledge the assistance provided by the school administration and staff. Their cooperation and provision of resources have been essential in the successful execution of this project.

Resumo

A visualização é uma ferramenta poderosa para explicar, compreender e depurar cálculos. Ao longo dos anos, várias ferramentas de visualização foram desenvolvidas para fins educacionais. A maioria dessas ferramentas alimenta os mecanismos de visualização usando os dados brutos do estado do programa disponíveis fornecidos pela API do depurador. Embora isso seja suficiente em certos contextos, há situações em que informações adicionais relevantes podem ajudar a criar visualizações mais abrangentes. Esta dissertação apresenta duas novas visualizações do Paddle, um ambiente de programação educacional baseado em informações sintetizadas de execução de programas. Geramos traços de execução e estados relevantes do programa através da análise estática e dinâmica dos dados de execução. As informações sintetizadas capturam comportamentos de programa que facilitam a criação de visualizações abrangentes e ricas envolvendo matrizes que descrevem leituras, gravações, movimentos e trocas de posições.

Foram realizadas entrevistas com o objetivo de elucidar as vantagens inerentes à aplicação Paddle. Cada entrevista era composta por quatro tarefas, cada uma das quais continha um erro no código que o entrevistado deveria identificar e subsequentemente corrigir. As entrevistas foram gravadas e o tempo decorrido até à identificação do erro e até à implementação da correção de código necessária foi documentado. Estes dados foram depois usados para facilitar a comparação entre a aplicação Paddle e o ambiente de desenvolvimento integrado (IDE) que os entrevistados estavam habituados a utilizar.

Palavras-chave: software educativo, linguagens de programação, educação de programação, visualizações de programação

Abstract

Visualization is a powerful tool for explaining, understanding, and debugging computations. Over the years, several visualization tools have been developed for educational purposes. Most of these tools feed visualization engines using the raw program state data available provided by the debugger API. While this suffices in certain contexts, there are situations where additional relevant information could aid in building up more comprehensive visualizations. This dissertation presents two novel visualizations of Paddle, an educational programming environment based on synthesized program execution information. We generate execution traces and relevant program states through static and dynamic analysis of the execution data. The synthesized information captures program behaviors that facilitate the creation of comprehensive and rich visualizations involving arrays that depict position reads, writes, moves, and swaps.

Interviews were conducted with the aim of elucidating the advantages inherent in the Paddle application. Each interview consisted of four tasks, each of which contained an error in the code that the interviewee had to identify and subsequently correct. The interviews were recorded, and the time taken to identify the errors and implement the necessary code correction was documented. This data was then used to facilitate a comparison between the Paddle application and the integrated development environment (IDE) that the interviewees were used to using.

Keywords: educational software, programming languages, programming education, programming visualizations

Contents

Figures index

Tables Index

List of Acronyms

API – Application Programming Interface

REST - Representational State Transfer

IDE – Integrated Development Environment

GUI – Graphical User Interface

UX/UI – User Experience/User Interface

OO – Object Oriented

AI – Artificial Intelligence

CHAPTER 1

Introduction

This dissertation begins with an examination of the contextual and motivational factors that shaped the research and development work conducted for this study. It then presents the methodology that was employed and provides an overview of the contributions of the work.

1.1 Context and Motivation

Programming educators commonly use illustrations to explain algorithms, in different forms, namely in their slides (possibly with animations), whiteboard explanations in the classroom, or on paper when addressing learners individually. Hence, program visualization tools appeal to many programming educators. However, a study [1] has shown that only about 20% of programming courses regularly use visualization tools and that almost half do not use them at all. The survey included responses from over 250 programming teachers and their students, who were asked about their use of visualization. Visualization tools are more often used by teachers working with younger students. The topics in which visualizations are most often used are introductory programming and data structures and algorithms.

Visualization tools are often integrated with debuggers or execution animators (e.g., [2], [3], [4], [5], [6]), where the tool renders the program state at each step. Except for PandionJ [4], these tools do not perform code analysis for capturing semantic aspects of the program (e.g., variable roles [7]) towards richer visualizations. The visualizations are often a mere alternative graphical representation of the information available in the call stack frames. Furthermore, debuggers do not provide the execution data regarding what happened before the program suspension at a breakpoint, making it difficult to illustrate the current program state in context. This leads to illustrations of program states that are less expressive than those hand-drawn by programming instructors [8], and the overall picture is lost through the debugging process.

1.2 Research Questions

The principal objective of this dissertation is to provide learners with a richer means to understand some programming basics and principles, such as recursion and expression resolution, and facilitate detailed observation of algorithmic behavior on arrays, including when errors occur. The objective is to ascertain whether a tool that can illustrate code information, such as the list of invocations that occurred during program execution and implement additional features comparable to those of other tools, as detailed in Section 2.3. This dissertation not only explores the feasibility of implementing a tool but also considers the impact on its target user base, namely students and instructors of introductory programming at the university level. To this end, the following research question is addressed:

RQ1: Is it feasible to implement a tool that illustrates the code execution?

RQ2: To what extent can the tool visualizations assist users in identifying and understanding program flaws?

1.3 Methodology and Contributions

This dissertation is based on the groundwork presented in the following paper [9]:

Educational Program Visualizations Using Synthetized Execution Information
Rodrigo Mourato, André L. Santos ICPEC'24, June 27-28, 2024, Lisbon, Portugal

As previously stated, the paper delineates the impetus and evolution of the developed tool, as outlined in Chapter 3. However, this dissertation provides a more comprehensive account of the tool's development and a user study that occurred after the paper was published.

In this dissertation, we describe automated program visualizations based on execution information synthesized from execution data, capturing traces and intents that are conventionally unavailable, such as expression-solving steps, array moves, and array swaps. When using our tool, users execute programs normally, and only if needed, may switch views to gain more execution insights without requiring specialized tool knowledge. We developed a web-based platform that supports a subset of Java, covering all the fundamental primitives for writing algorithms. We present two views with novel characteristics: (a) invocation tree with expression evaluation tracing; and (b) heap view with array history of reads and writes (capturing moves and swaps). These views aim to automate the hand-drawn illustrations of programming instructors using the results of a previous study [8]. In particular, the visualizations of array manipulations are novel concerning the state of the art, as we are unaware of any educational tool that illustrates moves and swaps explicitly (beyond depicting the raw program state step by step).

The process begins with the identification of the problem and the motivation for a novel solution. The objectives of this solution are then defined, allowing for the development of an initial prototype to demonstrate the practical feasibility of the proposed solution. The design and development process, which aims to provide an answer to research question RQ1, continues and is guided by an evaluation process through a user study, which provides an opportunity to answer research question RQ2. We start by identifying the problem and motivating the need for a novel solution, whose objectives are then defined so that an initial prototype can be developed to show the practical feasibility of the proposed solution.

1.4 Document Structure

This document commences with a literature review, which assesses and synthesizes the extant literature on the subject and its relationship to the present dissertation. The PRISMA methodology was selected for this review, and the findings are presented in the Data analysis section.

Following the literature review, the Paddle Environment section provides a detailed account of the tool's development, elucidating the rationale behind each illustration and the methodology employed in its implementation.

To assess the efficacy of our tool, a user study was conducted with 12 participants who were expected to possess a basic familiarity with Java. The findings are presented in Section 5.4, where times between two groups were calculated. This user study employed a within-subjects study design.

In the final section, the conclusions and recommendations for future work are presented. These include an analysis of the results from the user study as well as suggestions for improvements and features to enhance user comprehension of program execution.

CHAPTER 2

Literature Review

2.1 PRISMA

To conduct the literature review, the PRISMA methodology was chosen. This methodology implements guidelines for selecting which articles to include and exclude, considering the context of this dissertation and the availability of each article.

2.2 Query and Databases

The Scopus database was used to conduct the review using the query “educational” AND "programming languages" AND “views” in the search fields of article titles, abstracts, and keywords. Only articles and conference papers were included, resulting in 126 articles retrieved from Scopus. Additionally, 13 other articles were included in the study. After considering the context and content of each article, as well as their availability, 27 articles were included in the review.

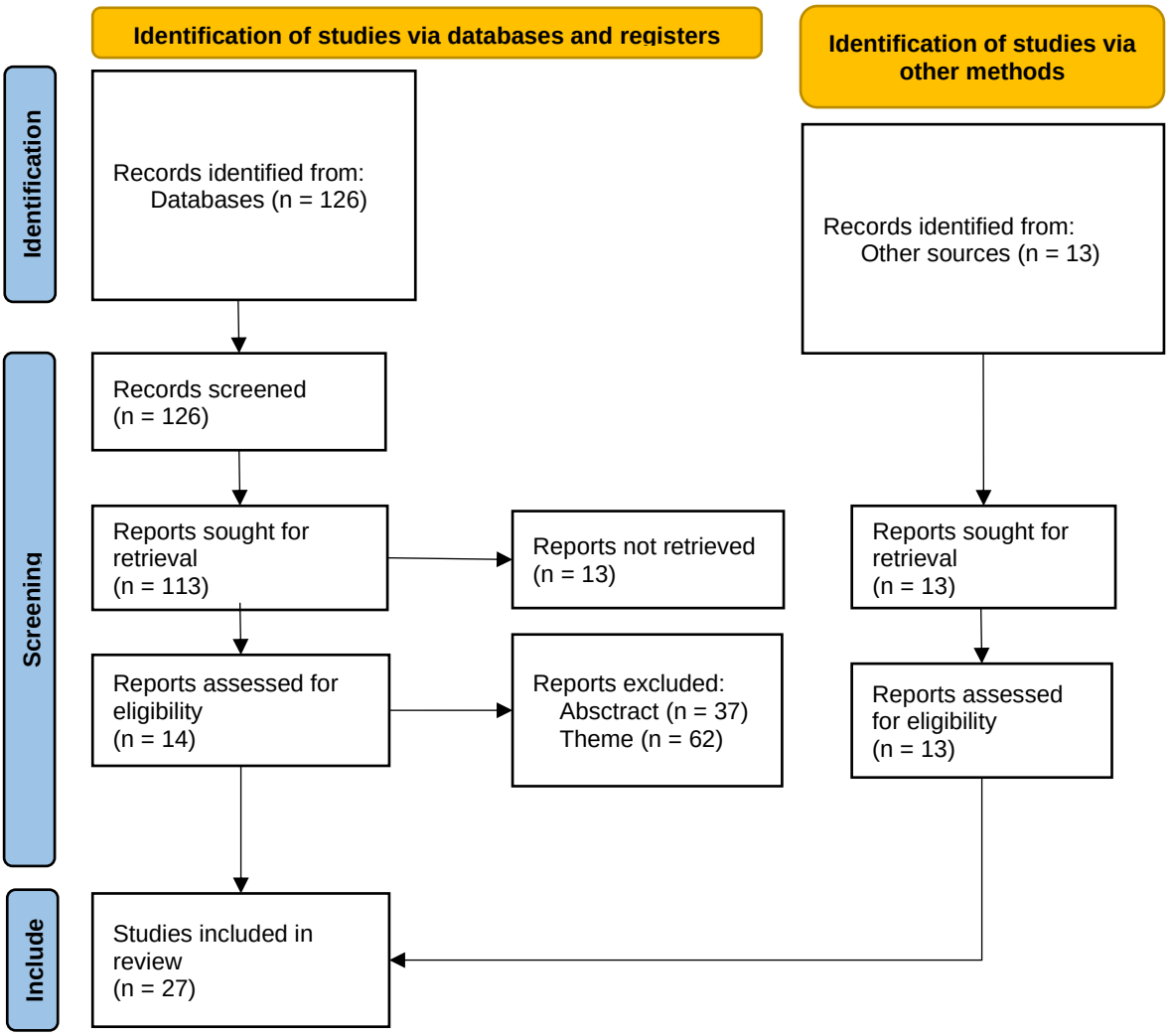

Figure 1 - PRISMA Flow Diagram [10]

All included articles underwent review and selection on September 16th, 2024. The selection criteria are presented in Figure 1, the PRISMA Flow Diagram.

2.3 Data analysis

When developing an application or a module with a Graphical User Interface (GUI), it is important to start with some basics. This includes architecture, distribution, navigation, color and text, equipment, values, tables, and alarms. It is crucial to investigate and analyze the best approaches for each of these aspects [11].

Surveys are a useful approach to gather feedback from students/learners and define the next steps. This may involve modifying the current GUI or implementing new features to enhance the learning process. According to a worldwide survey conducted by Essi Isohanni and Hannu-Matti Järvinen, only 20% of programming courses regularly use visualization tools, while slightly less than half do not use them at all. The survey included responses from over 250 programming teachers and their students, who were asked about their use of visualization [1]. Visualizations are more often used by teachers working with younger students. The topics in which visualizations are most often used are basic programming and data structures and algorithms.

2.3.1 Teaching and learning challenges

Teaching presents numerous challenges due to the vast number of topics covered and the need for students to comprehend all the information [12], [13]. Computer Science involves so many abstract concepts that can be challenging to grasp. As a course focused on practical applications, minimal reading materials and theory are provided. Daily tutorials are code-based, and assignments are language agnostic, providing most of the course content and expectations. Students are encouraged to regularly utilize search engines and are given tips and techniques to efficiently solve problems [12].

Java is a commonly used introductory programming course in Computer Science, but it is often considered challenging to teach and learn [14]. Students have identified the difficulty and attributed it to the object-oriented (OO) concepts and principles. To aid students, visualizations are a useful approach to be applied to programming environments [14].

To gain insight into the teaching and learning process, a survey was conducted in 2017. The survey employed four statements to gauge respondents' opinions. These statements included the following: “The experiment is better conducted in a group setting,” “The number of participants is sufficient,” “I prefer open-ended exercises (Lab 1),” and “I prefer step-by-step manual guidance (Lab 2)” [15]. All of the students who were surveyed indicated that the laboratory experiment should be conducted in a group of two with step-by-step procedures using the PDF.

In the realm of programming languages, some new approaches are being developed such as low code or block-based programming interfaces. Applications and frameworks that use this type of programming interfaces, such as Scratch, have emerged in the market and are being used as the basis for other projects, for example, to program a robot [16]. Tiled Grace is a new block-based programming interface, which was specifically designed for educational purposes at Victoria University of Wellington [17]. Finally, Pencil.cc is another project with educational purposes. Students who used code blocking instead of text, received higher assessment scores but lower confidence and enjoyment scores. Despite this, most of the students surveyed still plan to take more Computer Science courses after completing the course [18]. The Portugol IDE [19] is another example of a block-based programming interface, but in this case a Portuguese lexicon-based language is used to encode algorithms. Additionally, a new platform for teaching programming language syntax to beginners has been developed, inspired by educational techniques used to teach punctuation to children. The platform, called Hedy [20], begins as a basic programming language without any syntactic elements such as brackets, colons, or indentation. The rules gradually become more complex until the beginners are programming in Python. Hedy uses basic words such as “print”, “ask”, “echo”, “assign”, “assign list”, “if”, “else”, and “repeat”.

Social media has emerged as a valuable platform for learning programming. It offers access to code samples, applications, best practices, and advice. Recent studies have recognized social media as a valuable pedagogical tool for bridging the gap between formal and informal learning. TikTok is now being viewed as a platform for learning programming, potentially representing a new form of nano learning [21]. However, in contrast, Facebook has been used as a platform for enhancing the learning experiences of students in computer programming courses [22].

It is not uncommon for educators to overlook the fact that, from a student's perspective, the learning process at the university level can often be perceived as tedious and relentless. This is due to the fact that students are required to engage with a multitude of resources, including lectures and textbooks, which they must process and retain. This can result in information being received and subsequently forgotten rather than being fully integrated and retained [23].

2.3.2 Visualization tools

Software visualization includes two broad areas, algorithm visualization and program visualization, where the latter includes two further areas, visualization of static structures and visualization of runtime dynamics [24].

Algorithm visualization tools operate at a level of abstraction that is too high to be interesting for learning the basics of program execution. Prior to the students' engagement with a comprehensive IDE, they may benefit from an introduction to fundamental programming concepts through a basic framework. This could be complemented by a series of lessons on textual programming language training, providing structured training programs, exercises and online resoures before the commencement of the classes [25].

In terms of web-based tools, examples include VisuAlgo [26] and Algorithm Visualizer [27]. VisuAlgo is an online platform that offers interactive algorithm exercises, quizzes, and visualizations to aid in understanding common data structures and algorithms. However, it does not provide users with the ability to create their own visualizations. Algorithm Visualizer enables users to write code in multiple languages and visualize arrays, graphs, and individual values through a user-friendly GUI. Desktop tools such as LIVE [28], JFLAP [29], and JAWAA [30] are available. LIVE is a UML diagram generator that allows users to create UML diagrams from their own code. JFLAP is a graphical tool for creating visualizations of finite automata, Turing machines, and other constructs from automata and formal languages theory. Finally, JAWAA produces animations from code written in the JAWAA programming language. The project referred to in this article aims to be more flexible and capable of creating animations of arbitrary complexity [31].

UUhistle is a software tool designed to facilitate visual program simulation [5]. It provides graphical elements that students can manipulate to indicate what happens during execution, where, and when as shown in Figure 2. The tool only displays classes, functions, and operators that the program directly uses. Certain basic immutable data types have simplified default representations to maintain an organized visualization. UUhistle enables students to receive feedback on different types of errors, verify the accuracy of their answers, and obtain automated grading.

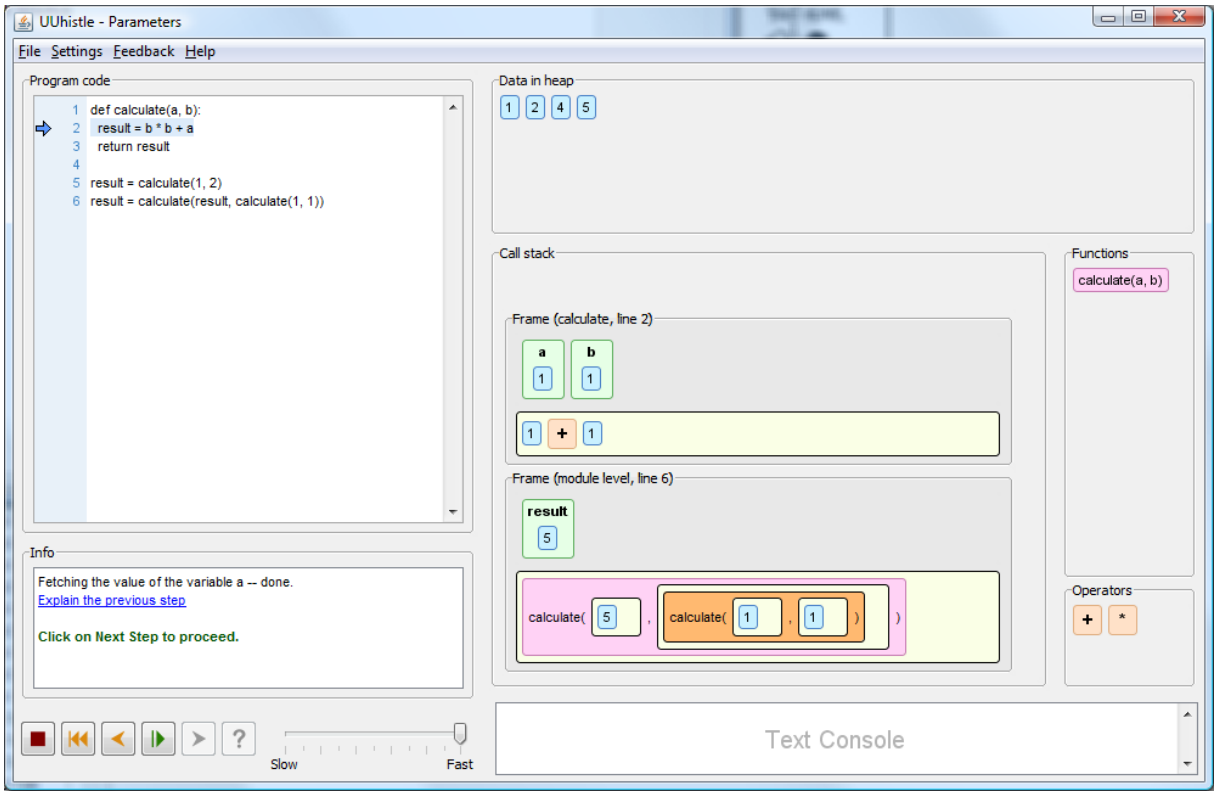

Figure 2 - UUhistle interface

JavlinaCode is a web-based programming environment that uses static and dynamic visualization to teach OO concepts [14]. The synchronized multi-view real-time visualization feature enhances the learning experience. The primary objective of this tool is to decrease the cognitive workload of students.

HDPV is a system for interactive, faithful, in-vivo runtime state visualization for native C/C++ programs and Java programs [32]. In the case of Java, the .java file is converted to a ByteCode file (.class) and then, using the JVM Runtime Instrumentation with their Java monitor (visasm), the information is synthesized and sent to the visualizer. The tool is designed to facilitate the use of multiple languages and to enable users to engage actively with their program's data set.

The SRec Visualization System [33] employs graphical representations to illustrate recursion trees as shown in Figure 3 - SRec interfaceFigure 3. Each node corresponds to a recursive call composed of two halves: the upper half contains the parameter values of the call, while the lower half contains the invocation’s result.

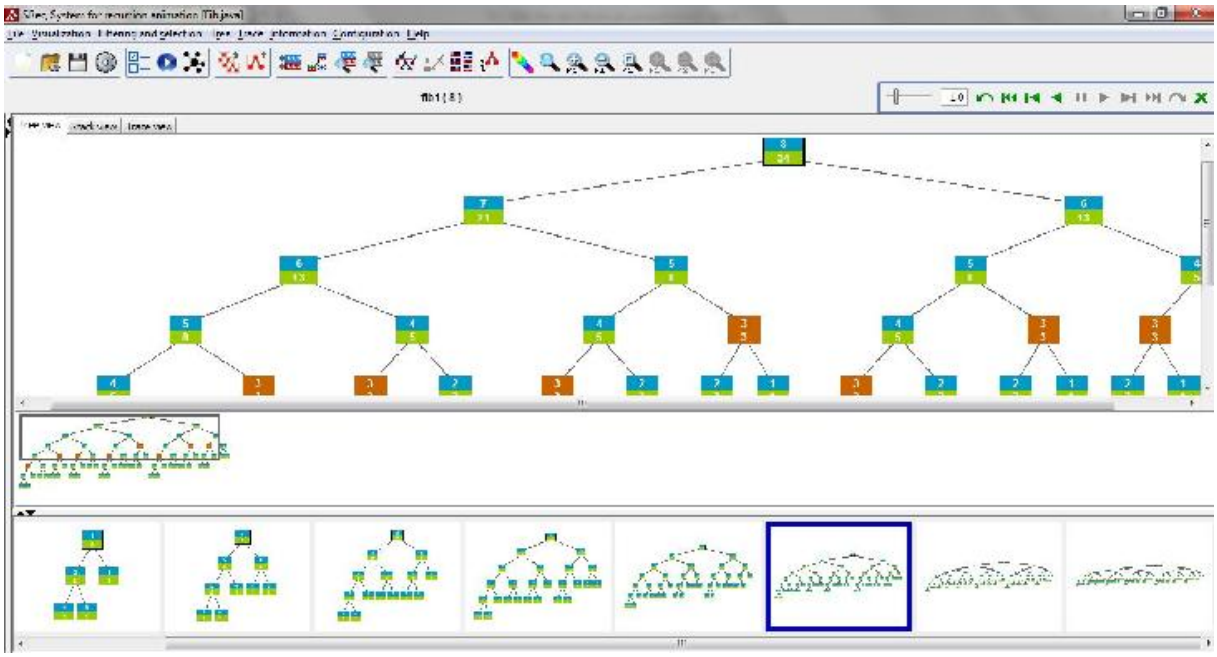

Figure 3 - SRec interface

WinHIPE [34] is an IDE for functional programming based on rewriting and visualization as shown in Figure 4. It also includes a powerful visualization and animation system that automatically generates visualizations and animations as a side effect of program execution.

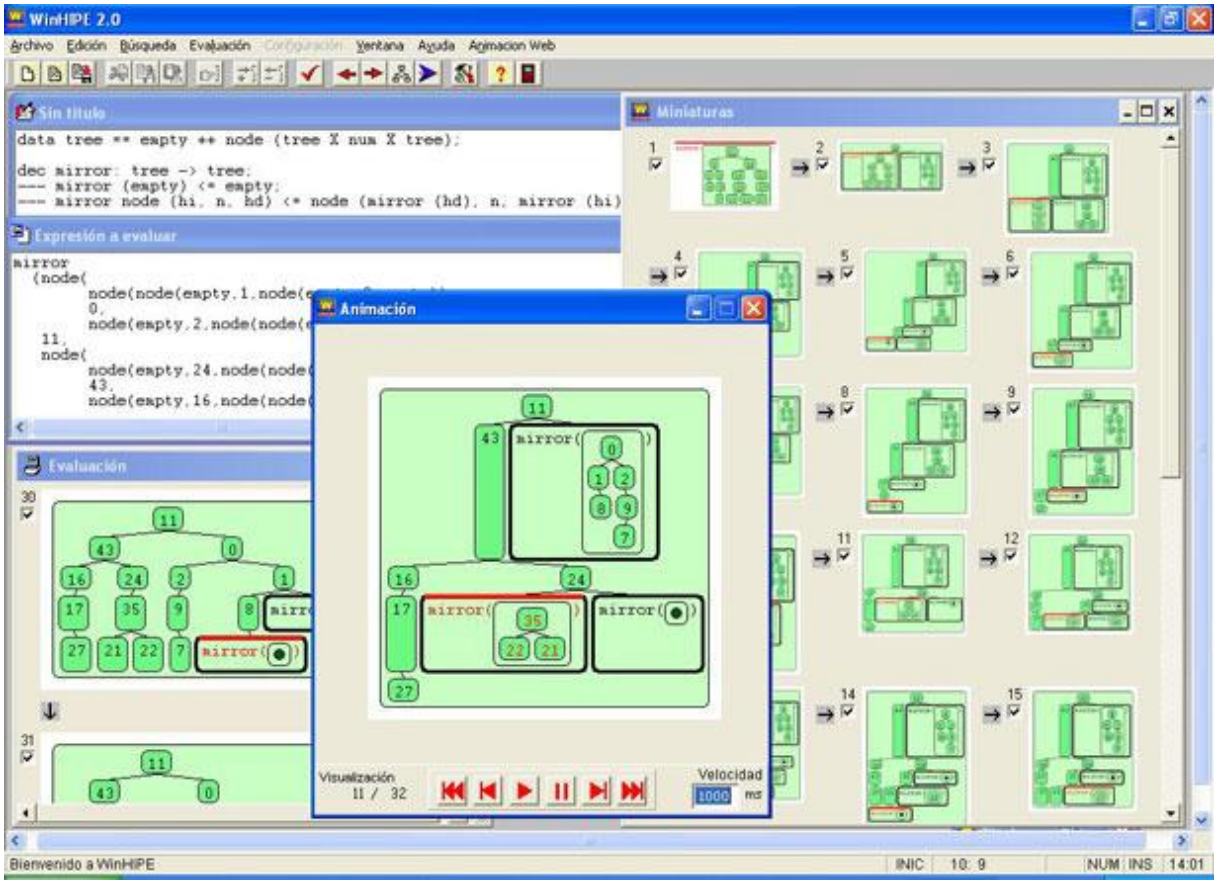

Figure 4 - WinHIPE interface

Introductory program visualization systems are often short-lived research prototypes that support user-controlled viewing of program animations [24]. Explanations in both OO and Artificial Intelligence (AI) courses are often accompanied by diagrams, figures, and visual aids [25].

2.3.3 Debuggers

Visualizations are often associated with debugging, whether it is static or dynamic. IDEs can utilize tools to provide declarative and visual debugging. For example, JIVE (Java Interactive Visualization Environment) is a declarative and visual debugging tool that has been integrated into the Eclipse IDE [2]. It was developed for educational purposes at the University of Buffalo. The authors concluded that JIVE is a lightweight tool that can be easily added to Eclipse. Performance is improved by updating the visualizations periodically at a user-defined interval.

Regarding debuggers with runtime visualizations, jGRASP is an IDE for visualizations to improve software comprehensibility. It is lightweight and provides static and dynamic visualizations of the user's program, and provides a conceptual rendering [3] as shown in Figure 5.

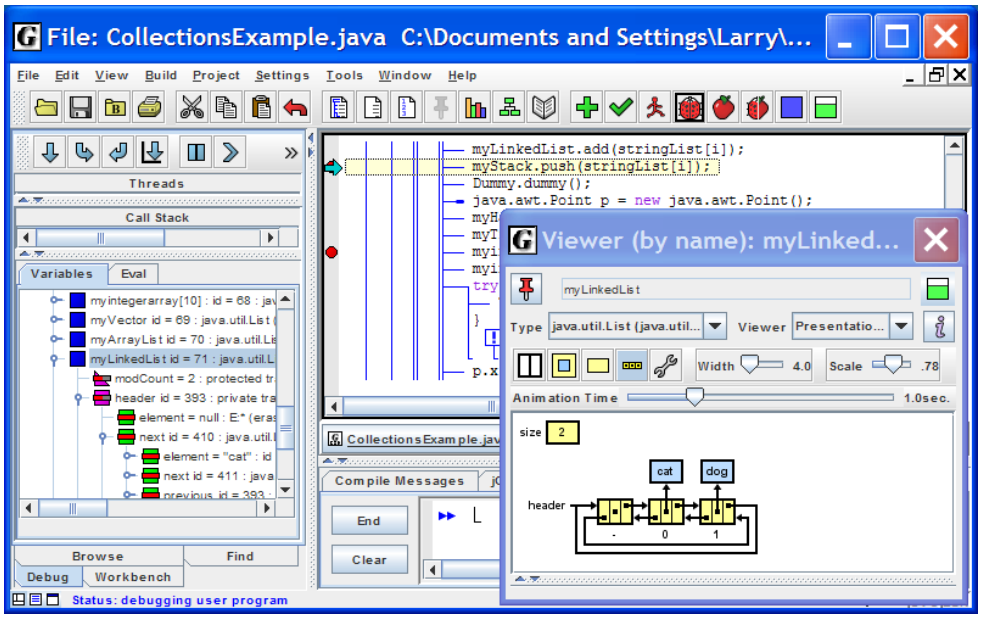

Figure 5 - jGrasp interface

A project of interest is Blink, an educational software debugger for Scratch [35]. It offers the useful feature of being able to navigate back in time to understand the differences between code before and after running each line. However, it should be noted that this project uses the block-based programming interface of the Scratch framework. This dissertation, on the other hand, requires users to develop their own code in Java, which is the programming language used in the introductory programming course at Iscte.

Another example is PandionJ [4], a pedagogical debugger for Java that is based on. It combines static analysis and graphical visualization as shown in Figure 6. Some visualizations were implemented, informed by a user study with programming instructors [8], namely accumulation terms trace (ATT), array index values (AIV), array index parameters (AIP), array index iterators (AII), search hit history (SHH), array index direction (AID), and array iteration bound (AIB). ATT is "writing the terms that make up an accumulation, leaving a trace of values that lead to the final result". AIV is “writing array indexes next to array locations”. AIP is “marking an array position whose index is given by a (fixed value) parameter”. AII is “writing iteration variables pointing to array positions/indexes”. SHH is “stroking the previous stored value during a search, leaving a trace of previous search hits that have been replaced by better values”. AID is “an arrow indicating the direction of an array index iterator (forward/backward)”. AIB is “a bar that divides an array according to the upper/lower bound of an array index iterator, typically in combination with the AID pattern”.

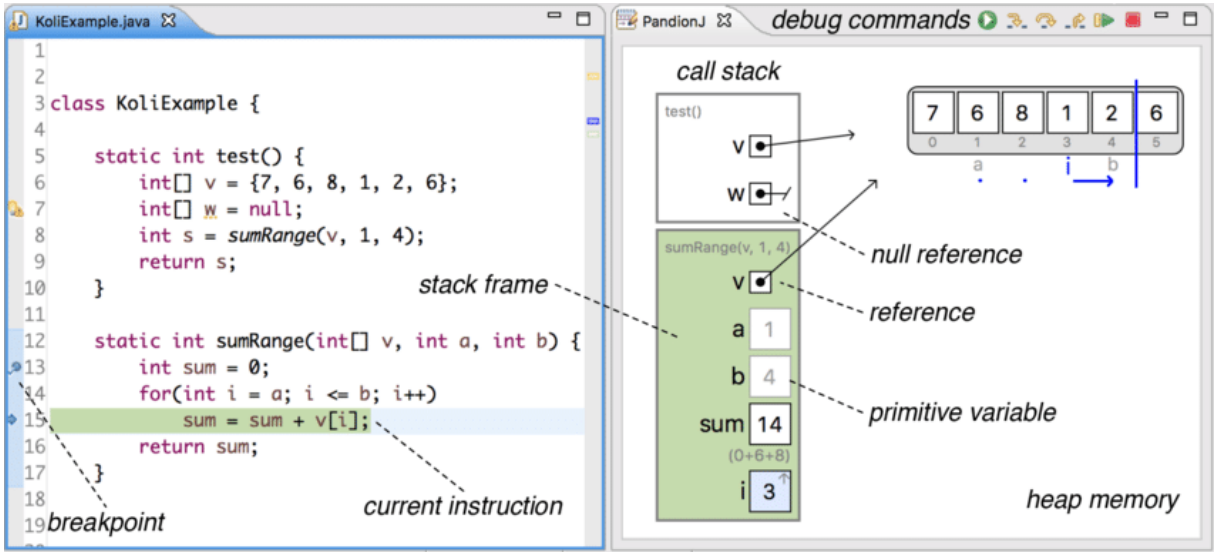

Figure 6 - PandionJ interface

Other examples of debuggers are BlueJ [36] and ViLLE [37]. BlueJ, a widely used programming environment, includes a debugger that works in the same style as a conventional debugger, but provides a simpler user interface aimed at beginners. ViLLE is a debugger that supports multiple programming languages.

Brown University developed Code Bubbles, a working-set based IDE that simplifies code display and navigation for the current task. The tool includes several advanced or experimental facilities and is under active development. Code Bubbles can display the debugging history as a UML sequence graph, the execution history of the current thread when it stops at a breakpoint, and information about a graphical user interface, including the widget hierarchy and the routines drawing at a selected pixel. It can also detail where the program is spending its time executing through a table showing the time spent at various lines and methods. The debugger should be able to detect and display detailed information about deadlocks when they occur. It should also display the value of programmer-defined expressions and update them at each breakpoint. Additionally, it should provide an interactive read-eval-print loop for the current context and a high-level view of the history of execution in terms of threads, tasks, and transactions. This view should be generated automatically based on data collected during previous debugging runs [38].

To trace and debug the program using jGRASP and other tools, an omniscient listener is required to provide information to users, which enables remote monitoring of program execution [39].

CHAPTER 3

Paddle Environment

Paddle is an innovative educational programming environment providing visualizations that leverage synthesized program execution information. It generates representative execution traces and relevant program states through static and dynamic analysis of the execution data.

The synthesized information captures diverse program behaviors to facilitate the creation of comprehensive and rich visualizations. The environment consists of a web application where the user can write code and obtain feedback about what happened during the execution as a trace illustration.

The user interface (UI) comprises two panels (Figure 7): the left panel, where the user writes code and executes programs, and the right panel, where alternative visualization panels are presented. Figure 8 illustrates the way how the user requests the code execution. When clicking the “Execute” button, a dialog prompts the user to enter the values for each parameter, and the current code is sent to the server with the specified function and arguments. Afterward, the code result is returned to the web application, and the user may check the outputs and switch among the available visualization panels, which we detail next. At the top of the right panel, the user has the option to switch between three views (Figure 7): the outputs view, which displays a list of outputs that were printed to the console; the invocation tree, which illustrates the list of invocations that occurred during program execution, showing the function name, parameters, and resolved return expression; and finally, the heap view, which illustrates the different array states that occurred during program execution, showing when an element was read, written, moved, or swapped.

Figure 7 - Paddle overview (switch between views)

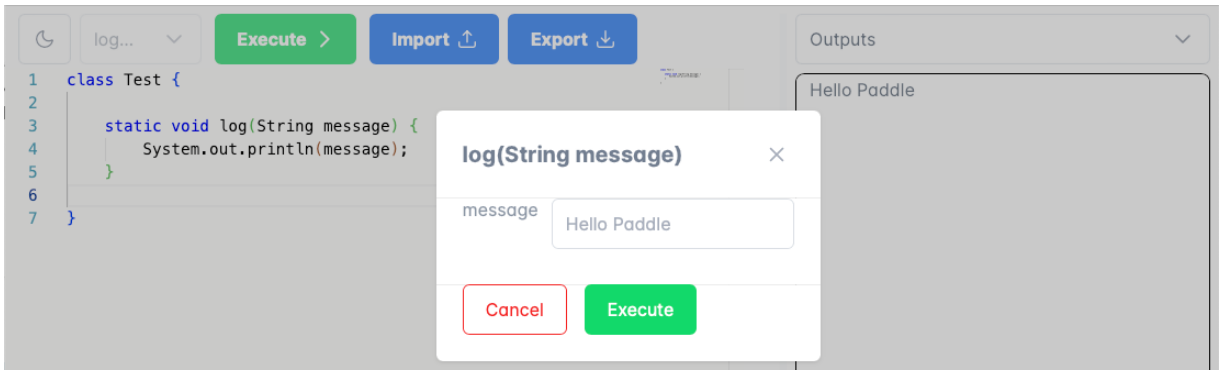

Figure 8 - Paddle environment: executing methods

3.1 Invocation Tree View

Figure 9 and presents a screenshot of the invocation tree view with the classic example of factorial calculation. Each node in the illustration represents one execution of a method, the solid edges represent invocations, and the dashed edges with the dashed nodes represent the return values of each invocation. If desired, the user may use the playback mode to go through each step, following the sequence of invocations. The related elements are selected in the code editor when clicking the view. When clicking an invocation node, the function declaration is highlighted, whereas when clicking a value node, the respective return expression is highlighted instead. This view is dynamic, allowing the user to click on the replay buttons to execute the view in a step-by-step manner, thereby facilitating an understanding of the manner in which the code was executed.

```
class Simple {
    static void log(String message) {
        System.out.println(message);
    }
    static int factorial(int n) {
        if(n == 0)
            return 1;
        else
            return n * factorial(n-1);
    }

    public static void main() {
        log("The factorial" +
          "of the number 3 is: " +
          factorial(3)
        );
    }
}
```

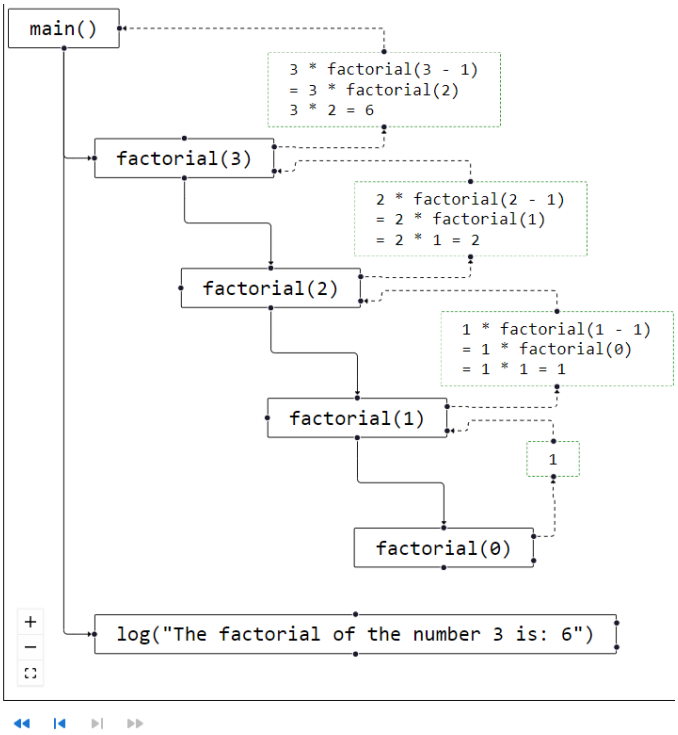

Figure 9 - Invocation tree view illustrating recursive calls (factorial calculation)

The main innovative feature of our view is the trace of expressions returned by the methods. In the example, the expression `3 * factorial(3 - 1)` is resolved to `3 * factorial(2)` and is finally resolved to `3 * 2`, which returns the final value of `6`. This enables the user to understand the return value of each invocation and how it was calculated. This information is synthesized from execution data and is not available when using debuggers (both educational and professional). For performance reasons, the total number of resolutions has a limit. Programming instructors often use similar illustrations to explain the execution of recursive calls [8].

3.2 Heap View

Figure 10 presents the heap view illustrating a function to check if an element is contained in an array. This view collects any array allocations performed in user code and renders its evolution through snapshots, from top to bottom. In this case, the array content remains the same because there are no side effects. The green background depicts that the highlighted position was read, whereas red denotes that a write operation was performed. In the illustration, we can observe that the last accessed position was the third one. The iterator variables for accessing array positions (*i* in the example) are depicted below the respective index (as in [4]). Programming instructors often use similar illustrations to explain computations that involve array iterations [8].

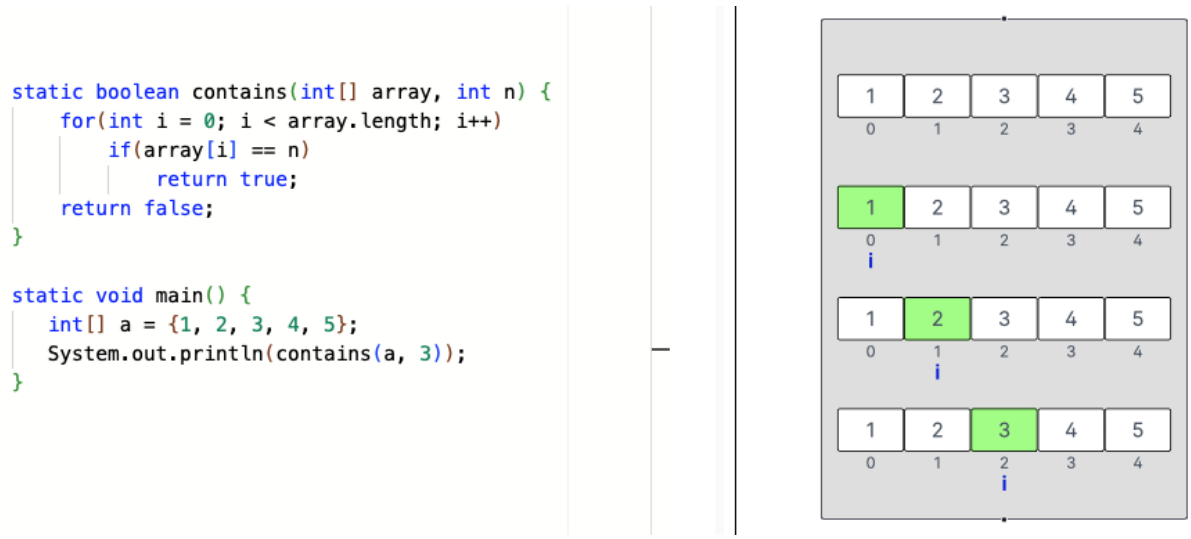

Figure 10 - Heap view illustrating array reads (check if element exists).

Figure 11 presents the heap view illustrating a procedure for left-shifting an array, exemplifying array writes. In the illustrations, a dashed arrow represents an array position move, that is, a value at one position is copied to another. This information is determined using a combination of static analysis and execution data.

```java
static void shift(int[] v) {
    int t = v[0];
    for(int i = 1; i < v.length; i++)
        v[i-1] = v[i];
    v[v.length-1] = t;
}
public static void main() {
    int[] array = {1, 2, 3};
    shift(array);
}
```

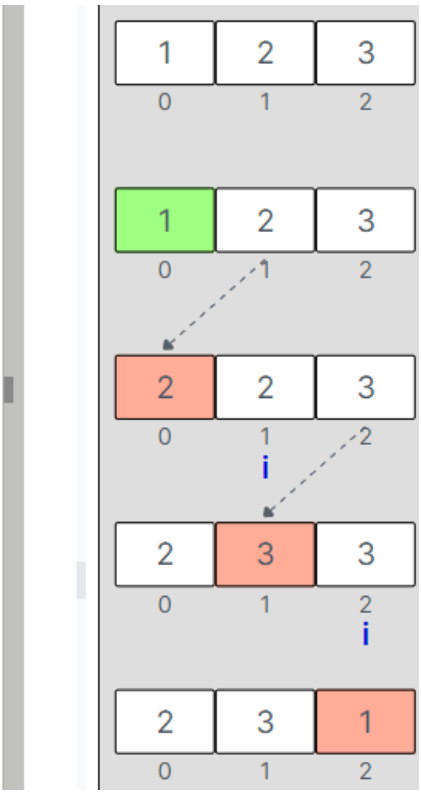

Figure 11 - Heap view illustrating array moves (left shift of array elements).

Figure 12 presents the heap view illustrating a procedure to reverse an array. The array was initialized with five elements and the reverse function was invoked, which internally invokes the function to swap two array elements given their indices. Special attention is paid to array swaps — information synthesized from execution data. As in array moves, a dashed arrow represents a move. Since a swap consists of two moves that exchange the values of the positions, the corresponding arrows are depicted simultaneously.

```java
static void swap(int[] a, int i, int j) {
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
}

static void reverse(int[] a) {
    int n = a.length;
    for (int i = 0; i < n / 2; i++) {
        swap(a, i, n - i - 1);
    }
}

public static void main() {
    int[] array = {1, 2, 3, 4, 5};
    reverse(array);
}
```

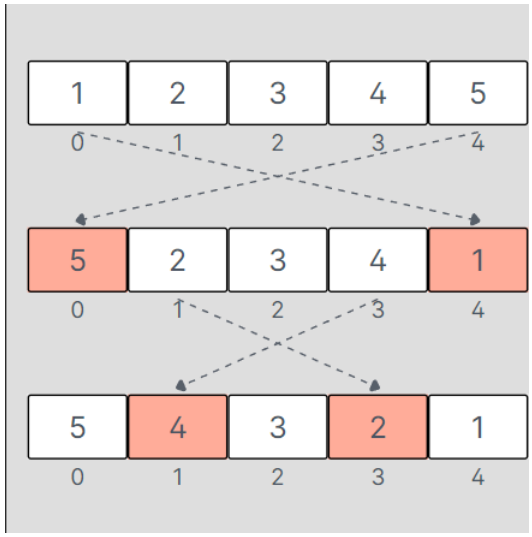

Figure 12 - Heap view illustrating array swaps (reverse the array)

If an array index out-of-bounds error occurs during execution, we illustrate the error in the view, as depicted in Figure 13. The expression that led to the invalid index is also marked with precision in the code. Recall that conventional support for this type of error typically consists of an error message that only includes the line number and invalid index (if multiple array accesses are in that line, the user must figure out which is causing the problem).

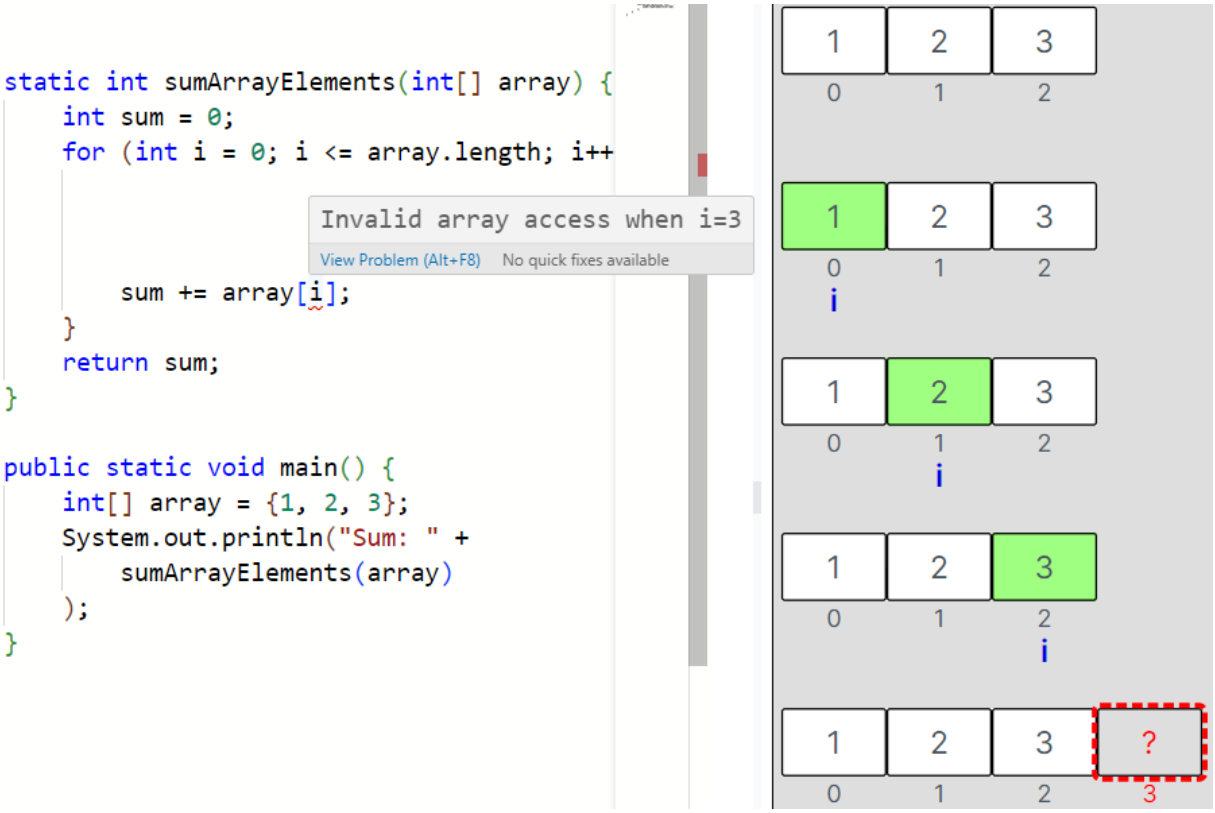

Figure 13 - Heap view illustrating an illegal access to an array position

CHAPTER 4

Implementation

The Paddle consist of a back-end information gathering and synthesis engine and a front-end interface for displaying the synthesized information with illustrations. The used frameworks are described in the following section.

4.1 Frameworks

Program execution and analysis are performed using Strudel[1], a programming library comprising classes that model structured programming, providing a virtual machine capable of interpreting those models, simulating the call stack-based execution. This enables clients to observe every aspect of execution in detail, including errors, tracking variable values, loop iterations, call stack, and memory allocation as shown in Figure 14. We developed execution listeners to gather the necessary information to render the views. Regarding the resolution of expressions, EvalEx[2] was employed. EvalEx is a convenient expression evaluator for Java that enables the parsing and evaluation of expression strings.

The back-end was constructed using Spring Boot[3], a JVM-based framework that simplifies the development of standalone application servers. The API calls respond JSON messages holding the execution results, outputs, traces, etc, that are necessary for building the visualizations.

The implementation of our prototype is based on a REST API, where program executions are performed, and a web-based front-end to display the results and visualizations. Ideally, the whole application could run on the browser, but we needed unavailable JavaScript libraries to execute the Java programs and synthesize the required information for the visualizations.

[1] https://github.com/andre-santos-pt/strudel
[2] https://github.com/ezylang/EvalEx
[3] https://spring.io/projects/spring-boot

The user interface was implemented using React[4], a popular JavaScript library for user interface development. The Redux Toolkit[5] was used for store management, providing utilities and abstractions to streamline common Redux tasks, such as creating actions, reducers, and store configuration. The code editor is provided by Microsoft Monaco[6], a lightweight, browser-based, highly versatile code editor providing features such as syntax highlighting, code completion, and IntelliSense. Monaco is the engine behind the Visual Studio Code editing experience and can be embedded in Web applications to edit code directly in the browser. Finally, the visualizations were implemented using React Flow library[7], a JavaScript library for developing interactive and visual flowcharts, diagrams, and graphs within React applications. It offers a flexible and customizable API to develop complex data visualization components, thereby enabling developers to incorporate drag-and-drop functionality, node-based layouts, and connection handling with relative ease. This library enabled the creation of custom nodes and edges, as illustrated in this paper's figures.

Figure 14 depicts the project's architectural components, wherein the front-end and back-end encompass the technologies. Upon clicking the "Execute" button, the user initiates the execution of the desired operation through the Rest API endpoint available in the Spring Boot application. Subsequently, the listener elements that were incorporated into the Strudel library are utilized to gather the requisite events, specifically "systemOutput", "procedureCall", "procedureEnd", "returnCall", "variableAssignment", "arrayAllocated", "elementChanged", and "elementRead". The EvalEx library is employed in the "returnCall" events to ascertain the results of the arithmetic expressions. Ultimately, the data is synthesized and conveyed to the front-end, which presents the information in a visually compelling manner.

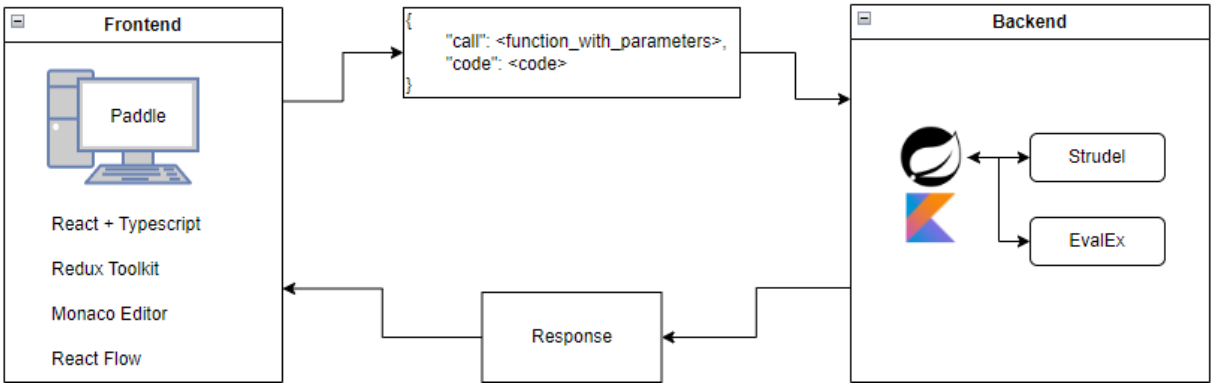

Figure 14 - Paddle environment architecture

[4] https://react.dev
[5] https://redux-toolkit.js.org
[6] https://microsoft.github.io/monaco-editor
[7] https://reactflow.dev

The response object is constituted by the principal function that the user intends to execute (root invocation), the enumeration of outputs, the enumeration of errors, and the enumeration of side effects. An invocation is a function that has been invoked and contains the following information: the function name, the function parameters, the function location, and the return values from the “Return Call”, which returns the result value, the function's return type, the result location, and the result calls that represent the list of resolved expressions. The outputs are the list of strings that were printed out to the console. Errors are represented by the list of errors that occurred during program execution. These include, for example, an invalid array access or a compilation error. Each invocation can have none or multiple calls. This represents invocations to other functions made by this one. Finally, the side effects field enumerates the side effects that occurred during the program's execution. These include alterations to a specific array, such as swaps, element reads or writes, and element moves. The interrelationship between these elements is illustrated in Figure 15 and discussed in greater detail in the subsequent section (Section 4.2).

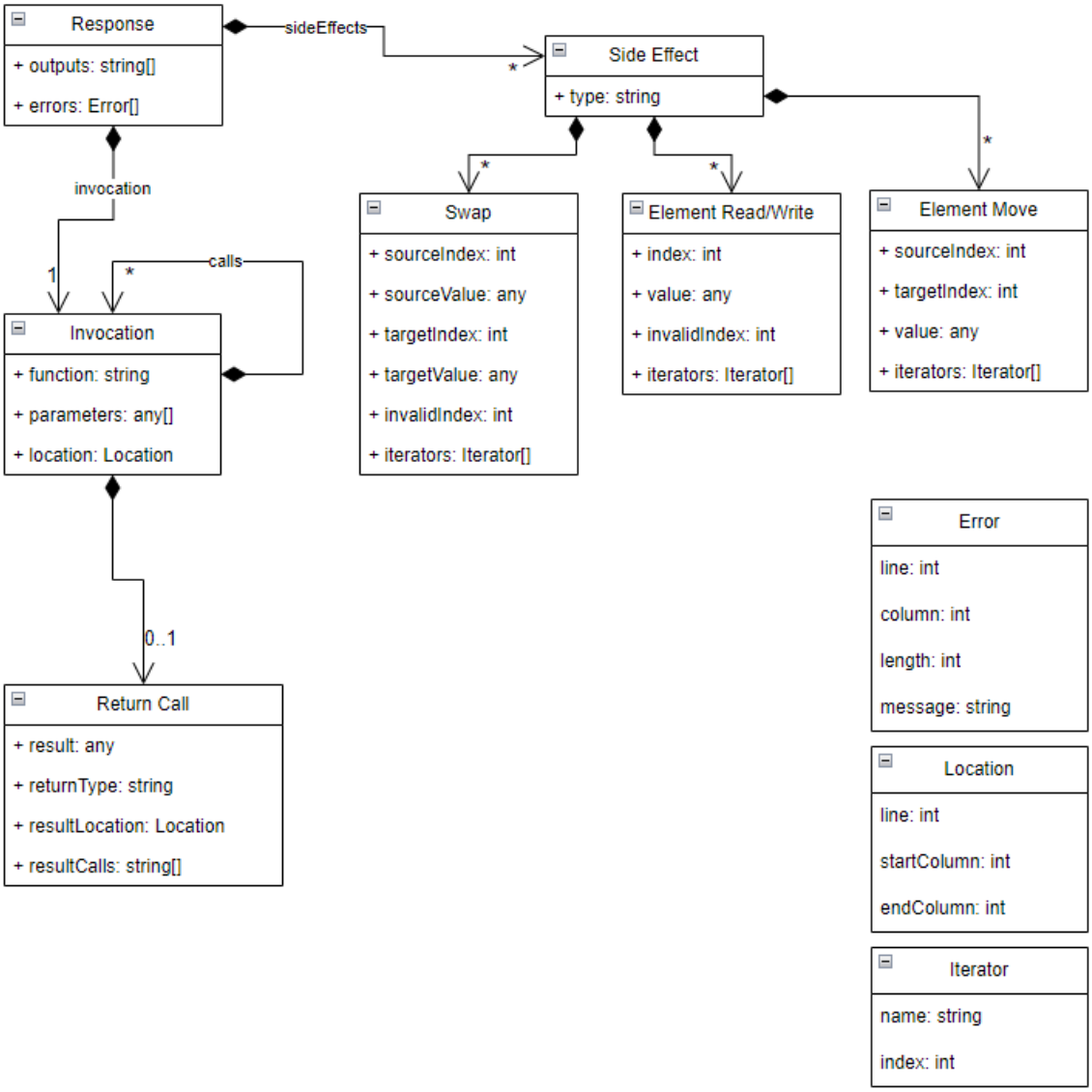

Figure 15 - Information collected and relations between listeners

4.2 Listeners

One of the initial implemented listeners was the one that captures procedure calls. It is beneficial to identify when an invocation is initiated and record it in the invocations list for presentation in the Invocation Tree view. The pseudo-code is illustrated below, and the pertinent code has been developed and is presented in Annex 1.

1. Extract procedure location:
    a. Get ID_LOC property of procedure.
    b. Convert it to a Location object.
2. Handle root invocation:
    a. If the caller is null:
        i. Update the location of the root invocation.
        ii. Set the root invocation return type to the procedure's return type.
        iii. Store root invocation frame with the top frame of the virtual machine.
        iv. Return.
3. Prepare caller details:
    a. Get the caller frame as the previous frame in the call stack.
    b. Serialize the caller frame arguments.
    c. Serialize the top frame arguments.
4. Create a new invocation with:
    a. The top frame of the virtual machine.
    b. An Invocation object with a unique ID, procedure ID, procedure call arguments values, location, and return type.
    c. An empty list of internal invocations.
5. Determine the parent invocation:
    a. If the caller frame is the root invocation, use the root invocation.
    b. Otherwise, find the matching invocation in root invocation’s internal invocations.
    c. Add the new invocation object to the parent's calls.
    d. Add the new invocation to the parent's internal invocations.

A supplementary listener has been devised to record the conclusion of an invocation. This listener is also utilized in the construction of the invocation tree view and is indispensable for the capture of the returned value associated with the preceding function. The pseudocode is presented below, with the corresponding implementation provided in Annex 2.

1. Serialize the result value.
2. Find the relevant invocation:

a. If the current frame is the root invocation:
    i. Use root invocation.
b. Otherwise:
    i. Find the invocation in root invocation's internal invocations that matches the current frame.

3. Update the invocation result:
    a. Store the invocation's result with the result value.

A further listener is required to construct the invocation tree view. This is triggered when a return statement is reached and captures the location of the return expression and the expression itself to be resolved. The following pseudo-code illustrates a listener that captures the return calls. The code in question has been developed and is presented in Annex 3.

1. Prepare variables:
    a. Clone JP property of return expression.
    b. Initialize result calls as an empty list of strings.
    c. Get the current frame from the virtual machine (top frame).
2. Update variable references in subs:
    a. For each variable reference in the return expression:
    b. Find the matching variable in current frame.
    c. Replace the variable reference name with its value (if it exists).
3. Update result calls:
    a. Add the returned and resolved expressions to the result calls.
    b. If the return value differs from the last result call, add it to the result calls.
4. Update invocation:
    a. Determine the invocation based on the current frame.
    b. Add the result calls to the invocation's result calls.
    c. Update the invocation's result location using the return expression location.

The addition of a listener to monitor all events associated with an array, whether an element is modified or accessed, to be used in the "heap view". The pseudocode is presented below, with the corresponding implementation provided in Annex 4.

1. Initialization:
    a. Store a copy of the initial array.
    b. Define variables track previous states.
2. Handle element changes:

a. Add a "write" side effect for the updated element.
b. If the new value matches the previous one:
    i. Remove the last three side effects (move, read, and write).
    ii. Add a "swap" operation between the previous and current indices.
    iii. Reset previous states.
c. Otherwise:
    i. Use the previously initialized array or the initial array.
    ii. If the new value exists in the array:
        1. Remove the last two side effects (read and write).
        2. Add a "move" side effect for the new element.
        3. Update previous states with the current ones.
    iii. Copy the updated array.

3. Handle element reads
    a. Add a "read" side effect for the accessed element.

4.3 Response Format

Regarding the response that is returned to the front-end, it respects the format specified in the next subsections. Each subsection describes a schema for a JSON object used in the response. The main object is the “Response”, which further decomposed in other objects (Table 1, Table 2, Table 3 and Table 4). For example, for the code executed in Figure 12 - Heap view illustrating array swaps (reverse the array), the response is the following one:


```
{
    "invocation": {
        "id": "e8e37c67-57e2-43d6-b3c7-efb564291b09",
        "function": "main",
        "location": {
            "line": 15,
            "startColumn": 24,
            "endColumn": 27
        },
        "calls": [
            {
                "id": "bccdfdb6-fcf3-44af-adff-85844945f289",
                "function": "reverse",
                "parameters": [
                    [
                        1,
                        2,
                        3,
                        4,
                        5
                    ]
                ],
                "calls": [
                    {
                        "id": "9970eaa8-d5d3-45d8-b316-b3894f3ce8a9",
                        "function": "swap",
                        "parameters": [
                            [
```

```
                                1,
                                2,
                                3,
                                4,
                                5
                            ],
                            0,
                            4
                        ],
                        "location": {
                            "line": 2,
                            "startColumn": 17,
                            "endColumn": 20
                        }
                    },
                    {
                        "id": "8656a517-de57-4862-a134-06e454d49113",
                        "function": "swap",
                        …
                    }
                ]
            }
        ]
    },
    "sideEffects": {
        "0": [
            {
                "id": "0a9c784c-61ff-49c4-8559-2ef30a159e6c",
                "type": "ARRAY_SWAP",
                "value": {
                    "array": [
                        5,
                        2,
                        3,
                        4,
                        1
                    ],
                    "sourceIndex": 0,
                    "sourceValue": 1,
                    "targetIndex": 4,
                    "targetValue": 5
                }
            }
        ]
    }
}
```


4.3.1 Response

Attribute	Description	Type
invocation	First invocation	Invocation
outputs	List of console outputs	String[]
errors	List of errors	Error[]
sideEffects	Map of side effects	{ arrayId: SideEffect[] }

Table 1 - Response Object

4.3.2 Invocation

Attribute	Description	Type
id	Invocation id that must be unique	String
function	Function's name	String
parameters	Function's parameters	Object[]
location	Function's location (where the function is declared)	Location
result	Function's result	Object
resultType	Specify the type of the result (string, double, float, etc)	String
resultCalls	List of strings with the resolved expressions	String[]
resultLocation	Specify where the result expression is declared	Location
calls	List of calls made by this invocation	Invocation[]

Table 2 - Invocation Object

4.3.3 Location

Attribute	Description	Type
line	Specifies the line number where the relevant piece of code is located	Integer
startColumn	Indicates the starting column number (or character position) on the specified line where the code fragment begins	Integer
endColumn	Marks the ending column number on the same line, showing where the code fragment finishes	Integer

Table 3 - Location Object

4.3.4 Error

Attribute	Description	Type
line	Specifies the line number where the relevant piece of code is located	Integer
column	Indicates the starting column number (or character position) on the specified line where the code fragment begins	Integer
length	Indicates the length of the error	Integer
message	Error message to be displayed	String

Table 4 - Error Object

4.3.5 Side Effect

Attribute	Description	Type
id	Side effect id	String
type	Specifies the side effect's type	String
value	Specified the details for this side effect (eg: source index, target index, iterators, etc.)	Any

Table 5 - Side Effect Object

CHAPTER 5

User Study

Following the completion of the Paddle prototype, a user study was conducted with the objective of elucidating the advantages inherent to the developed application. Prior to the commencement of the interviews, the participants were requested to consent to the audio, video, and screen recording of the interviews, which they all agreed to. It was required that the interviewees knew Java and had some experience with it.

5.1 Pilot interviews

Two pilot interviews were conducted, the interview script was revised and was then applied in 12 additional interviews. The script was adapted to align with the interviewers' perspectives, which allowed us to verify the IDE setup before starting the experiment and reinforce the fact that the participants could use console outputs to debug the program in both tools (IDE and Paddle) and run the program in debug mode when the task was being executed in IDE.

5.2 Tasks definition

Due to the low number of participants, we chose a within-subjects study design. Each participant had to accomplish four tasks, two of which were conducted using an IDE selected by the interviewer (control condition), and two other tasks were completed using the Paddle tool (experimental condition). The interviewers were divided into two groups and the tasks are presented in Table 6.

Task	Group 1	Group 2
Task 1	IDE	Paddle
Task 2	Paddle	IDE
Task 3	IDE	Paddle
Task 4	Paddle	IDE

Table 6 - Tasks order for each group

5.2.1 Task 1 – Factorial

Task 1 contains a recursive function often used in introductory programming, the factorial function. The function takes an integer as a parameter and recursively calculates the factorial of the number, returning the final result of the calculation. The code provides was the following:

```
1 | class Factorial {
2 |     static int factorial(int n) {
3 |         if(n == 0)
4 |             return 1;
5 |         else
6 |             return n + factorial(n-1);
7 |     }
8 |
9 |     public static void main(String[] args) {
10|         System.out.println("The factorial of the number 3 is: " + factorial(3));
11|     }
12| }
```

The error is present in line 6, where the return expression is wrong, because is applied a sum instead of a multiplication.

5.2.2 Task 2 – Shift right an array

Task 2 contains a function that takes an array as a parameter and moves the elements of the array one position to the right, with the last element at the first position. The code provided was the following:

```
1 | import java.util.Arrays;
2 |
3 | class ShiftRight {
4 |     static void shiftRight(int[] array) {
5 |         int last = array[array.length-1];
6 |
7 |         for(int i = array.length-2; i > 0; i--){
8 |             array[i + 1] = array[i];
9 |         }
10|
11|         array[0] = last;
12|     }
13|
14|     public static void main(String[] args) {
15|         int[] a = {1, 2, 3, 4, 5};
16|         shiftRight(a);
17|         System.out.println("The shifted right array is: " + Arrays.toString(a));
18|     }
19| }
```

The error is present in line 7, where the for condition is wrong because that causes a missing iteration, and it should be replaced with `i >= 0`.

5.2.3 Task 3 – Reverse an array

Task 3 contains a function that inverts an array, the code provided was the following:

```
1 | import java.util.Arrays;
2 |
3 | class Reverse {
4 |     static void swap(int[] a, int i, int j) {
5 |         int t = a[i];
6 |         a[i] = a[j];
7 |         a[j] = t;
8 |     }
9 |
10|     static void reverse(int[] a) {
11|         int n = a.length;
12|         for (int i = 0; i < n; i++) {
13|             swap(a, i, n - i - 1);
14|         }
15|     }
16|
17|     public static void main(String[] args) {
18|         int[] array = {1, 2, 3, 4, 5};
19|         reverse(array);
20|         System.out.println("The reversed array is: " + Arrays.toString(array));
21|     }
22| }
```

The error is in line 12, where the for condition is wrong, it should be replaced with `i < n/2`, because the for loop iterates the entire array instead of just half. At each iteration of the for loop, two elements are swapped, so we only need to iterate half of the array.

5.2.4 Task 4 – Sub array

The last assignment provided a function that creates a subarray from another, taking as input the array, the starting index, and the ending index. The indices are both inclusive. The code provided was as follows:

```
1 | import java.util.Arrays;
2 |
3 | class GenArray {
4 |     static int[] genArray(int[] a, int initial, int end) {
5 |         int[] newArray = new int[end - initial];
6 |         for (int i = 1; i <= newArray.length; i++) {
7 |             newArray[i - 1] = a[i + initial];
8 |         }
9 |         return newArray;
10|     }
11|
12|     public static void main(String[] args) {
13|         int[] array = {1, 2, 3, 4, 5};
14|         int[] subArray = genArray(array, 1, 4);
15|         System.out.println("The subarray is: " + Arrays.toString(subArray));
16|     }
17| }
```

This task contains three main problems, the size of the sub array that the expression should be replaced with `int[] newArray = new int[end - initial + 1];` and the for loop conditions that should be replaced with `for (int i = 0; i < newArray.length; i++) {` and finally the line 7 that is causing a negative index out of bounds and should be replaced with `newArray[i] = a[i + initial];`.

5.3 Participants characterization

Regarding the characterization of the participants, it can be stated that all the participants were male. With respect to age, 10 participants were below the age of 40, while 4 were aged 40 or above. About the distribution of professional experience, it can be observed that the percentage of younger and older individuals is comparable. The groups were constituted with members of varying ages to ensure equilibrium and to facilitate the examination of age-related differences. The participants were distributed according to their professional experience, with 28.6% having 0 – 3 years, 14.3% having 4 – 6 years, 14.3% having 7 – 10 years, 7.1% having 11 – 15 years, 28.6% having 16 – 20 years, and 7.1% having over 20 years, considering that both groups had the same number of elements with the same experience. As previously stated, all users had to be familiar with Java and possess at least a basic understanding of its functionality. An interval between 1 (very unfamiliar) and 5 (extremely familiar) was used to assess familiarity with the Java. Two of the 12 participants rated themselves at 2, three at 3, six at 4, and three at 5.

5.4 Results

The results are presented in Table 7, which provides a detailed overview of the minimum, maximum, and average times required to identify and resolve issues in each task. The bold numbers mark the fastest average times for each task (identification and fix).

Task	IDE (control)						Paddle (experimental)					
	Identification			Fix			Identification			Fix		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
1	00:25	01:00	**00:45**	00:10	00:45	**00:20**	00:25	03:00	01:10	00:05	03:35	01:00
2	00:20	02:50	01:50	00:05	01:10	**00:28**	00:20	03:40	**01:25**	00:20	02:10	01:11
3	02:00	06:40	04:05	00:05	01:40	00:56	00:45	03:05	**01:39**	00:10	01:25	**00:41**
4	00:20	02:15	01:11	01:15	03:35	02:39	00:25	01:40	**00:58**	01:00	04:40	**02:16**

Table 7 – Task completion times

The results demonstrated that Paddle can be a valuable tool for users in identifying and rectifying errors. However, the efficacy of this approach may not be as pronounced as desired. The mean time required to identify errors on Paddle is lower for three of the four tasks. The greatest reduction in time is observed for task 3, where participants required less than half the time taken on average in the IDE. The participants demonstrated a rapid and effective ability to identify and rectify errors in tasks 3 and 4. However, in the initial tasks, the participants required twice as much time as the IDE participants. This can be attributed to the initial interaction with the tool, as tasks 1 in Paddle was completed by Group 1 and task 2 in Paddle was completed by Group 2. The users evaluated and tested the tool, rather than proceeding directly to the code to resolve the error. However, this cannot be inferred from the data, as the users may have gained insights from the tool that facilitated the resolution process.

Regarding Task 1, the interviewers demonstrated a proclivity for utilizing the IDE in a prompt manner, subsequently transitioning to Paddle for both the identification and resolution phases. This phenomenon can be attributed to the initial encounter with a novel tool, which often entails a certain degree of hastiness. Additionally, among the six interviewers in Group 2 who completed Task 1 on Paddle, five lacked familiarities with the factorial expression. Conversely, only three of them exhibited this deficiency in Group 1.

Regarding Task 2, the participants demonstrated a greater ability to identify the issue within the Paddle than within the IDE tool. However, when attempting to resolve the identified problem, the participants exhibited a greater proficiency in the IDE tool than in the Paddle. Two of the six Group 1 (IDE) participants mentioned that the first element of the array had disappeared, whereas no one in Group 2 (Paddle) mentioned this, as the “for” loop was evident to be missing one iteration in Paddle.

Regarding Task 3, the interviewees utilizing Paddle were able to discern that the program was undergoing a swap operation twice, whereas the interviewers employing the IDE uniformly asserted that the program was not undergoing any such swap operation. This may account for the markedly superior performance of the Paddle users in comparison to those utilizing the IDE. Moreover, the Paddle users demonstrated a more expeditious resolution of the bug in comparison to the IDE users. Some users were observed to be investing a significant amount of time in identifying the issues. They were then asked to provide information regarding the number of swaps that should have occurred and the number of swaps that had occurred (two IDE users and one Paddle user).

About Task 4, it was not straightforward to calculate the time between identifying and solving the problems. Nevertheless, even if we add up the two times, the Paddle users were faster than the IDE users. Some users who were spending a considerable amount of time identifying the issues were asked about the size of the sub-array that was initially incorrect (one position was missing) to assist with identifying the underlying issues (one IDE user and one Paddle user).

5.5 Surveys

Following the conclusion of the interviews, the participants were requested to complete a survey. The survey was comprised of two sections: one pertaining to the participant and the other to the tool itself. Regarding to the developed application the list of questions is the following:

1. How would you rate the usefulness of the tool in general?
2. How would you rate the usefulness of the “Invocation Tree” view?
3. How would you rate the usefulness of the “Heap View” view?
4. How would you rate the usefulness of the tool for detecting errors or bugs?
5. How would you rate the tool's graphical interface?
6. What aspects of the tool do you find beneficial?
7. What are the tool's limitations?
8. Suggestions / Comments

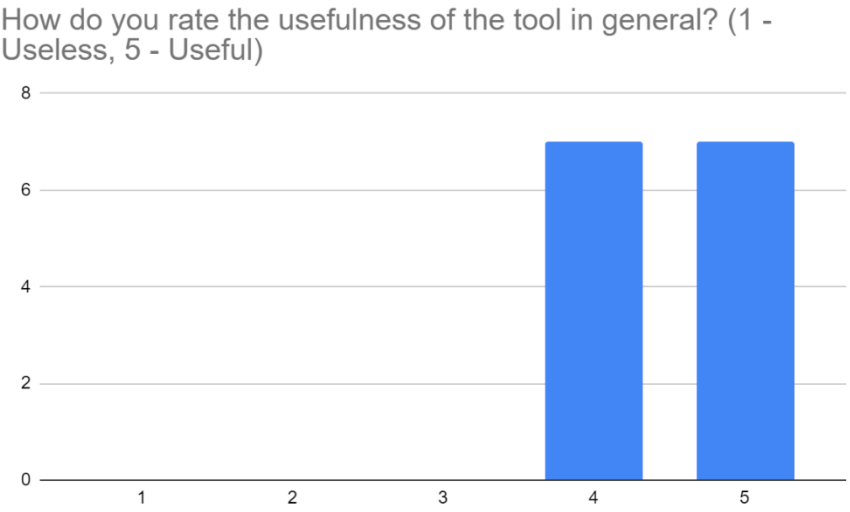

Figure 16 – How do you rate the usefulness of the tool in general?

In response to the question "How do you rate the overall usefulness of the tool?" (Figure 16) all respondents provided positive evaluations. Seven participants rated the tool's overall usefulness as 4, while the remaining seven rated it as 5, indicating that the tool is highly useful and can be used in a real-world setting.

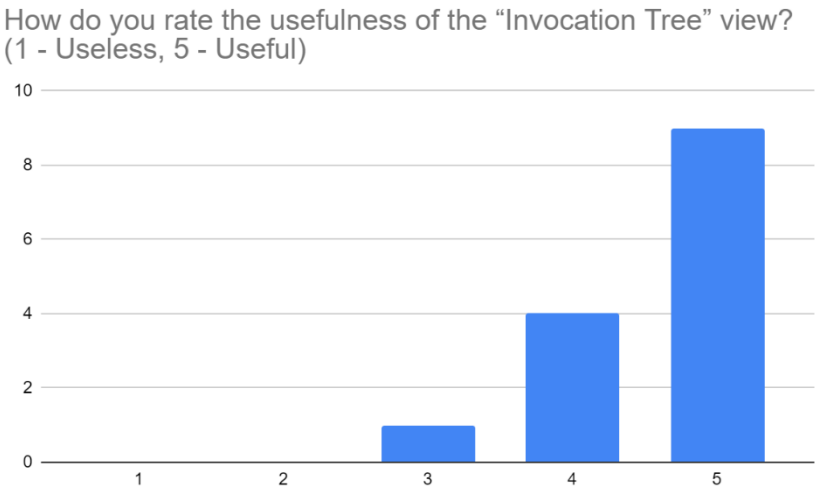

Figure 17 – How do you rate the usefulness of the "Invocation Tree" view?

Figure 17 illustrates the responses to the question "How do you rate the usefulness of the Invocation Tree view?". While most of the responses were positive, one answer was of intermediate rating, indicating a potential need for reevaluation of the illustration and identification of areas for improvement in this visualization.

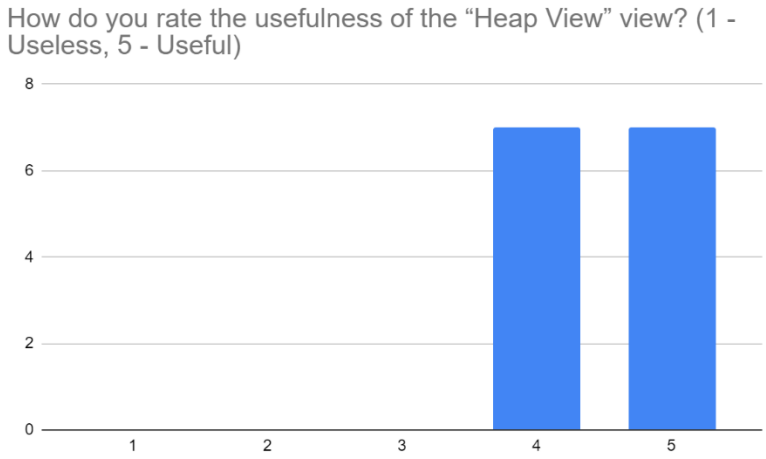

Figure 18 - How do you rate the usefulness of the “Heap View” view?

In response to the question "How do you rate the usefulness of the Heap View view?" (Figure 18), all respondents provided positive feedback, indicating that this view is highly effective in facilitating user comprehension of program execution outcomes. Additionally, during the interviews, most participants expressed positive sentiments regarding this view, further substantiating its well-designed nature.

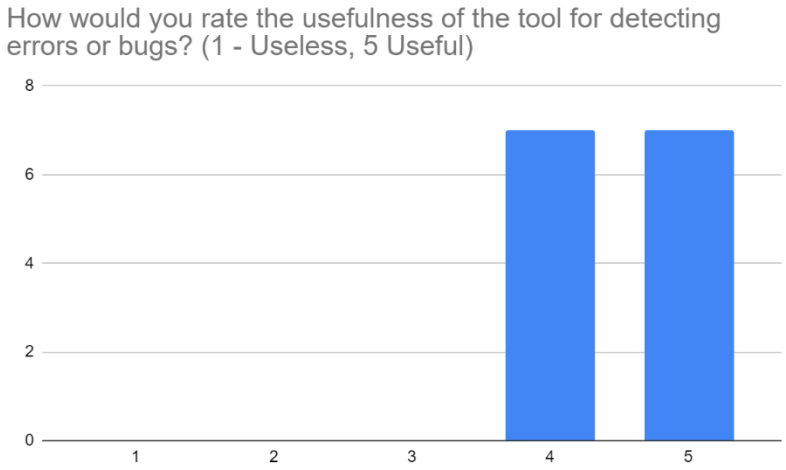

Figure 19 - How do you rate the usefulness of the tool for detecting errors or bugs?

In response to the question “How do you rate the usefulness of the tool for detecting errors or bugs?” (Figure 19), all respondents provided positive feedback, indicating that the errors that are presented to the users are being useful. However, the tool does not handle all errors, which represents a potential area for improvement.

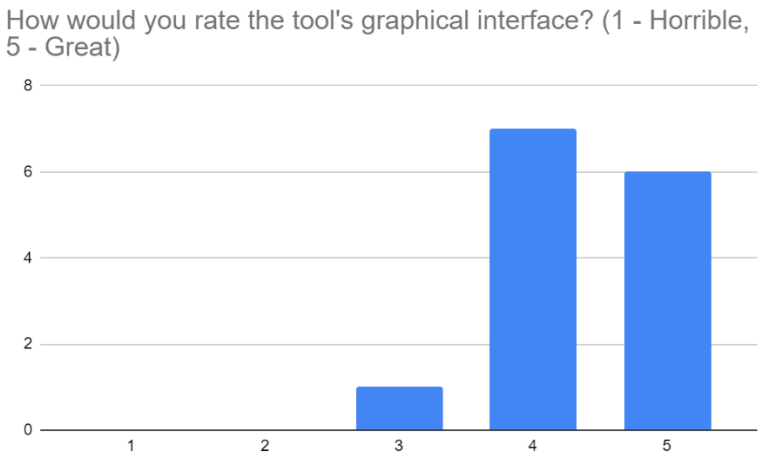

Figure 20 - How would you rate the tool's graphical interface?

Regarding the user experience (UX) and user interface (UI) aspects, the question “How would you rate the tool's graphical interface?” was posed, and 13 responses were positive, while one was intermediate (Figure 20). This indicates that most users are satisfied with the UX/UI, but there is one user who is less satisfied. These findings suggest that further improvements can be made and that additional experimentation with a larger user base is necessary.

To gain further insight into the perceptions of the tool, users were requested to identify the advantages (Table 8) and constraints (Table 9) of the tool and to provide suggestions and/or comments (Table 10).

The fact that you can visualize what is happening is very good and can visually help someone to identify an error more easily.
Graphical interface for debugging
It makes it easier to understand code execution and detect bugs
Being able to get the results of the execution graphically
The graphical demonstration (of both the Heap View and the Invocation Tree) of the code to serve as a teaching support tool for users learning Java or for debugging for users with more or less experience.
Graphical debugging helps to better understand the various iterations of the code.
The biggest benefit would be that it makes it easier for people who are "new" to Java programming to learn and familiarize themselves with the concepts. Moreover, even for people with some experience, it can help you find bugs or notice anomalous behavior in your application.
Graphical visualization of data; in the case of the heap exercise, the sequence of execution was much clearer.
Possibility of graphically visualizing objects.
In some problems, being able to visualize the execution and see the change in the code reflected in this change was very good.
Visual representation of the execution of each instruction, which helps and makes the debugging process much easier.
Having a teaching tool that provides a graphical view of what is happening is very useful, I think that for those who are starting to program, seeing what is happening simplifies the learning process and allows the student to be focused on achieving the solution of the exercise. The tool also indicates where the source of the error might be, which also allows the student to be more autonomous
It seems very useful as a learning tool at an introductory level of programming
Being able to visualize all invocations and changes without having to go to one and one in the debugger

Table 8 - Please describe the advantages of the tool in question

In terms of the advantages of the tool, most of the identified questions are already incorporated into other tools. However, it is encouraging to note that the users can discern these features and recognize the tool's utility. For instance, most of the users indicated that the views facilitate comprehension of code execution.

Clicking on the arrays always went to the initialization of the array and not where that change happened
Only one file
It took me a while to understand how to interpret the invocation tree.
The biggest limitation is that it is not possible to use other programming languages. This tool with support for low-level languages such as C would be very useful as it is one of the most widely used languages for introducing people to the world of programming.
I don't know if it would be as easy in more complex environments
I think that the color representation (green, red) of what is being done in the heap view, and even to access that same option is not very intuitive, I don't think it will be a major limitation, and maybe it's just because it's the first contact.

Table 9 - What are the constraints of the tool?

In terms of constraints, the users provided valuable insights. One concern that was identified was regarding the tool's capacity to handle more complex environments. One limitation of the tool was identified as its inability to support the manipulation of multiple files simultaneously. However, this is not considered a significant limitation, given the intended use of the tool in introductory programming classes. As previously stated, a user encountered difficulties in comprehending the invocation tree. About the UX/UI, the users observed that the color representation for reads and writes (green and red) lacks intuitive clarity. Therefore, it is recommended to consider this aspect and implement improvements in the future. Regarding user interaction, the users noted that a constraint exists in Heap View when a node is clicked. Instead of indicating the modification that occurred, the view always points to the array initialization.

Extend the application so that it can support more programming languages.

Table 10 - Suggestions / Comments

In conclusion, about the suggestions and comments that were provided, it should be noted that only one comment was made, which was addressed in the constraints section. It is inaccurate to state that the application "only" supports Java. In this dissertation, Java was the primary language of focus. However, the front-end is processing a JSON object that is formatted in a specific structure previously mentioned in this document. If another server were available that could handle events such as array modifications and return a JSON object with analogous data, it would be displayed in a similar manner. Moreover, the editor being utilized (Monaco) already supports a range of languages beyond Java, making it potentially more straightforward to adapt it to be dynamic and choose by the user.

CHAPTER 6

Conclusion and future work

Our prototype demonstrates that rich program visualizations can be obtained in a post execution manner by making use of synthetized execution information. Our visualizations are inspired by illustrations often made by programming instructors (e.g., in slides, animations, or hand-drawn). In particular, the array manipulation illustrations are unavailable in other visualization tools supporting arbitrary user code, and without having to execute the program step-by-step (as when using a debugger). We argue that our views are a quick means to illustrate the execution of simple programs involving invocations and arrays, with minimal need to learn any specific tool features.

Regarding the user study, the participants demonstrated that the tool is a valuable asset in the identification and resolution of bugs. The participants found the heap view to be more useful than the invocation tree view. However, only one task directly interacts with the invocation tree view, while the other three interact with the heap view. There was a notable difference in the time required to complete the tasks with the control and experimental tools. Nevertheless, we believe that the identification time is more crucial than the fix time. Allowing users to interact with the tool for a longer duration allows them to gain a deeper understanding of the illustration, which is the primary focus of this dissertation. This, in turn, enables them to become more proficient and expedient in identifying and resolving the underlying issues in the code. The user study contributes to answering the research questions by demonstrating the feasibility of implementing a tool that illustrates code execution and assists users in identifying and understanding program flaws through the use of tool visualizations.

As future work, we plan to evaluate how programming instructors perceive the usefulness of our visualizations. Evaluating the tool from the perspective of programming beginners could also inform how easily and accurately they interpret the visualizations. Even if the visualizations have no expressive effect on novices working autonomously, they may serve as an aid to instructors when assisting learners in lab classes or remotely, sparing time that otherwise would be spent on figuring out what went wrong with the program execution and manually drawing illustrations for further explanations.

Regarding tool improvements, we plan to support objects in the heap view, which are important to illustrate elementary data structures such as linked lists and trees and to elaborate on the illustrations of errors (e.g., stack overflows). A further evaluation of the views and tool UX/UI is recommended, with testing conducted with a larger sample size. As previously stated by Jakob Nielsen, testing with a mere five users will likely reveal most usability issues, while testing with six or more users may yield diminishing returns in terms of new insights. The primary conclusion is that a relatively small group of users can effectively identify a substantial proportion of usability issues, with approximately five to six users detecting about 80% of an application's problems [40]. Furthermore, we believe that more interactivity between the views and the source code could improve the user experience, and we acknowledge that strategies to cope with large drawings are necessary for good usability.

Bibliography

[1] E. Isohanni and H.-M. Järvinen, “Are visualization tools used in programming education?: by whom, how, why, and why not?,” in *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Finland: ACM, Nov. 2014, pp. 35–40. doi: 10.1145/2674683.2674688.

[2] J. K. Czyz and B. Jayaraman, “Declarative and visual debugging in Eclipse,” in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, Montreal Quebec Canada: ACM, Oct. 2007, pp. 31–35. doi: 10.1145/1328279.1328286.

[3] J. Cross, D. Hendrix, L. Barowski, and D. Umphress, “Dynamic program visualizations: an experience report,” in *Proceedings of the 45th ACM technical symposium on Computer science education*, Atlanta Georgia USA: ACM, Mar. 2014, pp. 609–614. doi: 10.1145/2538862.2538958.

[4] A. L. Santos, “Enhancing Visualizations in Pedagogical Debuggers by Leveraging on Code Analysis,” in *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, Koli Finland: ACM, Nov. 2018, pp. 1–9. doi: 10.1145/3279720.3279732.

[5] J. Sorva and T. Sirkiä, “UUhistle: a software tool for visual program simulation,” in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Finland: ACM, Oct. 2010, pp. 49–54. doi: 10.1145/1930464.1930471.

[6] R. Ben-Bassat Levy, M. Ben-Ari, and P. A. Uronen, “The Jeliot 2000 program animation system,” *Computers & Education*, vol. 40, no. 1, pp. 1–15, Jan. 2003, doi: 10.1016/S0360-1315(02)00076-3.

[7] J. Sajaniemi, “An empirical analysis of roles of variables in novice-level procedural programs,” in *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, Arlington, VA, USA: IEEE Comput. Soc, 2002, pp. 37–39. doi: 10.1109/HCC.2002.1046340.

[8] A. L. Santos and H. Sousa, “An exploratory study of how programming instructors illustrate variables and control flow,” in *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, Koli Finland: ACM, Nov. 2017, pp. 173–177. doi: 10.1145/3141880.3141892.

[9] R. Mourato and A. L. Santos, “Educational Program Visualizations Using Synthetized Execution Information,” 2024.

[10] “PRISMA.” Accessed: Feb. 03, 2024. [Online]. Available: http://prisma-statement.org/prismastatement/flowdiagram.aspx?AspxAutoDetectCookieSupport=1

[11] F. Rodríguez, J. L. Guzmán, M. Castilla, J. A. Sánchez-Molina, and M. Berenguel, “A proposal for teaching SCADA systems using Virtual Industrial Plants in Engineering Education,” in *IFAC-PapersOnLine*, Elsevier B.V., 2016, pp. 138–143. doi: 10.1016/j.ifacol.2016.07.167.

[12] S. Qiu, F. Zhang, and Z.-Y. Peng, “Design a game of charged particles moving in a uniform magnetic field based on unity 3D-Take ‘the wise snake through the pass’ as an example,” in *ACM Int. Conf. Proc. Ser.*, Association for Computing Machinery, 2021, pp. 12–16. doi: 10.1145/3474995.3474998.

[13] T. Xiao, R. I. Greenberg, and M. V. Albert, “Design and Assessment of a Task-Driven Introductory Data Science Course Taught Concurrently in Multiple Languages: Python, R, and MATLAB,” in *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, Virtual Event Germany: ACM, Jun. 2021, pp. 290–295. doi: 10.1145/3430665.3456364.

[14] J. Yang, Y. Lee, D. Hicks, and K. H. Chang, “Enhancing object-oriented programming education using static and dynamic visualization,” in *Proc. Front. Educ. Conf. FIE*, Institute of Electrical and Electronics Engineers Inc., 2015. doi: 10.1109/FIE.2015.7344152.

[15] A. Jaafar, N. Soin, and S. W. M. Hatta, “An educational FPGA design process flow using Xilinx ISE 13.3 project navigator for students,” in *Proc. - IEEE Int. Colloq. Signal Process. Appl., CSPA*, Institute of Electrical and Electronics Engineers Inc., 2017, pp. 7–12. doi: 10.1109/CSPA.2017.8064915.

[16] P. Plaza *et al.*, “Build your own robot,” in *IEEE Global Eng. Edu. Conf., EDUCON*, Klinger T., Kollmitzer C., and Pester A., Eds., IEEE Computer Society, 2021, pp. 543–551. doi: 10.1109/EDUCON46332.2021.9453965.
[17] M. Homer and J. Noble, “Combining tiled and textual views of code,” in *Proc. - IEEE Work. Conf. Softw. Vis., VISSOFT*, Sahraoui H., Zaidman A., and Sharif B., Eds., Institute of Electrical and Electronics Engineers Inc., 2014, pp. 1–10. doi: 10.1109/VISSOFT.2014.11.
[18] D. Kopetzki, M. Lybecait, S. Naujokat, and B. Steffen, “Towards language-to-language transformation,” *Int. J. Softw. Tools Technol. Trans.*, vol. 23, no. 5, pp. 655–677, 2021, doi: 10.1007/s10009-021-00630-2.
[19] A. Manso, C. G. Marques, and P. Dias, “Portugol IDE v3.x: A new environment to teach and learn computer programming,” in *IEEE EDUCON 2010 Conference*, Madrid: IEEE, Apr. 2010, pp. 1007–1010. doi: 10.1109/EDUCON.2010.5492469.
[20] F. Hermans, “Hedy: A Gradual Language for Programming Education,” in *Proceedings of the 2020 ACM Conference on International Computing Education Research*, Virtual Event New Zealand: ACM, Aug. 2020, pp. 259–270. doi: 10.1145/3372782.3406262.
[21] M. B. Garcia, I. C. Juanatas, and R. A. Juanatas, “TikTok as a Knowledge Source for Programming Learners: A New Form of Nanolearning?,” in *Int. Conf. Inf. Educ. Technol., ICIET*, Institute of Electrical and Electronics Engineers Inc., 2022, pp. 219–223. doi: 10.1109/ICIET55102.2022.9779004.
[22] Ö. Özyurt and H. Özyurt, “Using Facebook to enhance learning experiences of students in computer programming at Introduction to Programming and Algorithm course,” *Comput Appl Eng Educ*, vol. 24, no. 4, pp. 546–554, 2016, doi: 10.1002/cae.21730.
[23] S. Street and A. Goodman, “Some experimental evidence on the educational vahxe of interactive Java applets in Web-based tutorials”.
[24] J. Sorva, V. Karavirta, and L. Malmi, “A Review of Generic Program Visualization Systems for Introductory Programming Education,” *ACM Trans. Comput. Educ.*, vol. 13, no. 4, pp. 1–64, Nov. 2013, doi: 10.1145/2490822.
[25] W.-Y. Lu and S.-C. Fan, “Developing a weather prediction project-based machine learning course in facilitating AI learning among high school students,” *Comput. Educ.*, vol. 5, 2023, doi: 10.1016/j.caeai.2023.100154.
[26] “visualising data structures and algorithms through animation - VisuAlgo.” Accessed: Feb. 13, 2024. [Online]. Available: https://visualgo.net/en
[27] “Algorithm Visualizer,” Algorithm Visualizer. Accessed: Feb. 13, 2024. [Online]. Available: https://algorithm-visualizer.org/
[28] A. E. R. Campbell, G. L. Catto, E. E. Hansen, and H. College, “Language-Independent Interactive Data Visualization”.
[29] “JFLAP.” Accessed: Feb. 13, 2024. [Online]. Available: https://www.jflap.org/
[30] “The JAWAA HomePage.” Accessed: Feb. 13, 2024. [Online]. Available: https://www2.cs.duke.edu/csed/jawaa2/
[31] D. Jeffries, R. Mohan, and C. Norris, “DsDraw: Programmable animations and animated programs,” in *ACMSE - Proc. ACM Southeast Conf.*, Association for Computing Machinery, Inc, 2020, pp. 39–46. doi: 10.1145/3374135.3385292.
[32] J. Sundararaman and G. Back, “HDPV: interactive, faithful, in-vivo runtime state visualization for C/C++ and Java,” in *Proceedings of the 4th ACM symposium on Software visualization*, Ammersee Germany: ACM, Sep. 2008, pp. 47–56. doi: 10.1145/1409720.1409729.
[33] J. Á. Velázquez-Iturbide and A. Pérez-Carrasco, “How to use the SRec visualization system in programming and algorithm courses,” *ACM Inroads*, vol. 7, no. 3, pp. 42–49, Aug. 2016, doi: 10.1145/2948070.
[34] C. Pareja-Flores, J. Urquiza-Fuentes, and J. Á. Velázquez-Iturbide, “WinHIPE: an IDE for functional programming based on rewriting and visualization,” *SIGPLAN Not.*, vol. 42, no. 3, pp. 14–23, Mar. 2007, doi: 10.1145/1273039.1273042.

[35] N. Strijbol, R. De Proft, K. Goethals, B. Mesuere, P. Dawyndt, and C. Scholliers, “Blink: An educational software debugger for Scratch,” *SoftwareX*, vol. 25, 2024, doi: 10.1016/j.softx.2023.101617.
[36] M. Kölling, “The BlueJ system and its pedagogy,” *Computer Science Education*, Dec. 2003, Accessed: Feb. 13, 2024. [Online]. Available: https://www.academia.edu/2657382/The_BlueJ_system_and_its_pedagogy
[37] T. Rajala, M.-J. Laakso, E. Kaila, and T. Salakoski, “VILLE – A Language-Independent Program Visualization Tool”.
[38] S. P. Reiss, “The Challenge of Helping the Programmer during Debugging,” in *2014 Second IEEE Working Conference on Software Visualization*, Victoria, BC, Canada: IEEE, Sep. 2014, pp. 112–116. doi: 10.1109/VISSOFT.2014.27.
[39] K. Shimari, T. Ishio, T. Kanda, and K. Inoue, “Near-Omniscient Debugging for Java Using Size-Limited Execution Trace,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, OH, USA: IEEE, Sep. 2019, pp. 398–401. doi: 10.1109/ICSME.2019.00068.
[40] “Why You Only Need to Test with 5 Users.” Accessed: Aug. 30, 2024. [Online]. Available: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/

CHAPTER 7

Attachments

Annex 1 - Call listener to capture invocations

```
override fun procedureCall(procedure: IProcedureDeclaration, args: List<IValue>, caller: IProcedure?) {
    val locationProperty = procedure.getProperty("ID_LOC")!! as SourceLocation
    val location = Location(locationProperty.line, locationProperty.start, locationProperty.end)
    if (caller?.id == null) {
        rootInvocation.invocation.location = location
        rootInvocation.invocation.returnType = procedure.returnType.toString()
        rootInvocation.frame = virtualMachine.topFrame
        return
    }
    val callerFrame = virtualMachine.callStack.previousFrame!!
    val procedureCallArgsValues = args.map { serializeValue(it.value) }
    val newInvocation =
        InternalInvocation(
            virtualMachine.topFrame,
            Invocation(
                UUID.randomUUID().toString(),
                procedure.id!!,
                procedureCallArgsValues,
                location,
                result: null,
                procedure.returnType.toString()
            ),
            mutableListOf()
        )
    val invocation =
        if (isRootInvocation(rootInvocation, callerFrame)) rootInvocation
        else findInvocation(rootInvocation.internalInvocations, callerFrame)
    invocation?.invocation!!.calls.add(newInvocation.invocation)
    invocation.internalInvocations.add(newInvocation)
}
```

Annex 2 - Listener to capture the end of the invocation

```
override fun procedureEnd(procedure: IProcedureDeclaration, args: List<IValue>, result: IValue?) {
    val resultValue = serializeValue(result?.value)
    val invocation =
        if (isRootInvocation(rootInvocation, virtualMachine.topFrame)) rootInvocation
        else findInvocation(rootInvocation.internalInvocations, virtualMachine.topFrame)!!
    invocation.invocation.result = resultValue
}
```

Annex 3 - Listener to capture return calls

```
override fun returnCall(s: IReturn, returnValue: IValue?) {
    val subs = (s.expression?.getProperty("JP") as Expression).clone()
    val resultCalls: MutableList<String> = mutableListOf()
    val currentFrame = virtualMachine.topFrame
    subs.findAll(NameExpr::class.java).forEach { varRef ->
        val dec = currentFrame.variables.keys.find { it.id == varRef.nameAsString }
        val value = currentFrame.variables[dec]
        value?.let {
            varRef.setName(it.toString())
        }
    }
    resultCalls.add(subs.toString())
    resultCalls.addAll(resolveExpression(rootInvocation, subs.toString()))
    val lastResultCall = resultCalls.last()
    if (returnValue.toString() != lastResultCall) {
        resultCalls.add(returnValue.toString())
    }
    val invocation =
        if (isRootInvocation(rootInvocation, currentFrame)) rootInvocation
        else findInvocation(rootInvocation.internalInvocations, currentFrame)!!
    invocation.invocation.resultCalls.addAll(resultCalls)
    val resultLocation = s.expression?.getProperty(SourceLocation::class.java)
    invocation.invocation.resultLocation =
        Location(resultLocation!!.line, resultLocation.start, resultLocation.end)
}
```

Annex 4 - Strudel listener to capture array operations

```
override fun arrayAllocated(ref: IReference<IArray>) {  new *
    ref.target.addListener(object : IArray.IListener {
        val initialArray = ref.target.copy()
        lateinit var prevArray: IArray
        var prevIndex: Int? = null
        var prevOld: IValue? = null

        override fun elementChanged(index: Int, oldValue: IValue, newValue: IValue) {
            sideEffects.putMulti(
                ref.target, SideEffect(
                    UUID.randomUUID().toString(),
                    SideEffectType.ARRAY_ELEMENT_WRITE, ReadWriteSideEffect(
                        (serializeValue(ref.target) as MutableList<*>).toMutableList(),
                        index,
                        serializeValue(newValue.value),
                         invalidIndex: null,
                        getProcedureIterators()
                    )
                )
            )
            if (newValue == prevOld) {...} else {
                val array = if (::prevArray.isInitialized) prevArray else initialArray
                if (array.elements.indexOf(newValue) > -1 ||
                    (array == initialArray && initialArray.elements.indexOf(newValue) > 1)
                ) {
                    val arraySideEffects = sideEffects[ref.target]!!
                    // Remove last 2 operations (Read and Write)
                    arraySideEffects.removeAt( index: arraySideEffects.size - 1)
                    arraySideEffects.removeAt( index: arraySideEffects.size - 1)
                    sideEffects.putMulti(
                        ref.target, SideEffect(
                            UUID.randomUUID().toString(),
                            SideEffectType.ARRAY_ELEMENT_MOVE, ArrayElementMove(
                                (serializeValue(ref.target) as MutableList<*>).toMutableList(),
                                array.elements.indexOf(newValue),
                                index,
                                serializeValue(newValue.value),
                                getProcedureIterators()
                            )
                        )
                    )
                }
                prevIndex = index
                prevOld = oldValue
            }
            prevArray = ref.target.copy()
        }

        override fun elementRead(index: Int, value: IValue) {
            sideEffects.putMulti(
                ref.target, SideEffect(
                    UUID.randomUUID().toString(),
                    SideEffectType.ARRAY_ELEMENT_READ, ReadWriteSideEffect(
                        (serializeValue(ref.target) as MutableList<*>).toMutableList(),
                        index,
                         value: null,
                         invalidIndex: null,
                        getProcedureIterators()
                    )
                )
            )
        }
```