INSTITUTO
UNIVERSITÁRIO
DE LISBOA

An Educational Environment for Code Behaviour Inspection

Afonso Maria Pissarra Mendonça Centeno Neves

Master's in Computer Science and Engineering

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

Co-Supervisor:
PhD Sancho Moura Oliveira, Associate Professor,
Iscte - Instituto Universitário de Lisboa

October, 2024

TECHNOLOGY
AND ARCHITECTURE

Department of Information Science and Technology

An Educational Environment for Code Behaviour Inspection

Afonso Maria Pissarra Mendonça Centeno Neves

Master's in Computer Science and Engineering

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

Co-Supervisor:
PhD Sancho Moura Oliveira, Associate Professor,
Iscte - Instituto Universitário de Lisboa

October, 2024

Acknowledgments

I would like to thank everyone that contributed to the development of this project and helped me create a product I can be proud of presenting.

First, I would like to acknowledge my supervisors, André L. Santos, who proposed the topic of this dissertation and provided guidance throughout, and Sancho M. Oliveira, whose feedback and advice were instrumental in shaping the project's design.

I am also thankful for my family and friends for their support and encouragement throughout the development of this dissertation. I would like to thank my parents, who have always provided me with the opportunities to achieve the best possible outcome in all aspects of my life; Sofia, whose constant presence and care supported me through every stage of this project; and Afonso, who generously went above and beyond to greatly enhance the quality of the final work.

Finally, I would like to thank the anonymous volunteers that kindly agreed to participate in our study and contribute their time and insight.

Resumo

A programação é uma disciplina inerentemente difícil de aprender e ensinar devido à sua natureza abstrata e complexa. Alunos em cursos introdutórios de programação enfrentam regularmente desafios significativos que dificultam o seu percurso de aprendizagem desde cedo. É também comum professores manifestarem dificuldades em ajudar os alunos com os seus problemas de programação, uma vez que identificar a origem de um problema no código nem sempre é trivial. Debuggers pedagógicos têm demonstrado melhorar a experiência académica para ambos, porém, as abordagens existentes geralmente requerem uma determinada proficiência em conceitos de programação e, frequentemente, operam num ambiente separado do espaço dedicado à escrita de código.

Esta dissertação apresenta um protótipo de ambiente de debugging para Java integrado no espaço de desenvolvimento de código, permitindo uma transição harmoniosa entre a escrita e inspeção comportamental do código. O ambiente permite aos utilizadores aceder a históricos de variáveis durante uma execução, consultar informação relativa a ciclos, avaliar expressões em diferentes momentos do programa (nomeadamente, condições e operações numéricas), e examinar mensagens de apoio para tratamento de erros. Além disso, o ambiente visa promover a prática de test-driven development, fornecendo uma estrutura para testes unitários integrada no sistema de debugging.

Realizámos um estudo para avaliar a eficácia do protótipo no auxílio à resolução de problemas de programação. Recrutámos voluntários inexperientes na área, com o objetivo de simular a experiência de um aluno principiante. Concluímos que o protótipo se mostrou útil numa variedade de problemas que poderiam ser propostos a iniciantes em ambientes académicos.

Palavras-Chave: *Programação introdutória, inspeção de comportamento, debugger pedagógico*

Abstract

Programming is an inherently difficult subject to learn and teach due to its complex and abstract nature. Students in introductory programming courses often face severe challenges which hinder their learning path at an early stage. Teachers also recurrently struggle to assist students with their coding issues, as identifying the source of a problem is not always straightforward. Pedagogical debuggers have been shown to enhance the experience for both, however, existing approaches generally require a certain level of programming expertise to be used effectively and often function in a setting separate from the coding workspace.

This dissertation presents a prototype of a debugging environment for Java that is integrated with the coding area, allowing for a seamless transition between the tasks of writing and inspecting code. These inspections allow users to examine variable histories, gather information on loop iterations, evaluate expressions at different points in the code (namely, conditions and numerical operations), and examine error handling support messages. In addition, the tool aims to promote test-driven development by providing a structure for writing unit tests that are fully integrated into the debugging system.

We conducted a study to evaluate the effectiveness of the prototype in assisting users in solving programming problems. For this purpose, we recruited inexperienced volunteers simulating the experience a beginner student would have when using the tool. We concluded that the prototype proved useful across a range of problems that could be presented to novice students in an academic setting.

Keywords: *Introductory programming, behaviour inspection, pedagogical debugger*

Contents

List of Figures

List of Tables

List of Acronyms

IDE: Integrated Development Environment

TDD: Test-Driven Development

SDK: Software Development Kit

UI: User Interface

UML: Unified Modeling Language

GUI: Graphical User Interface

API: Application Programming Interface

CHAPTER 1

Introduction

This chapter introduces the topic of our work, stating the motivation behind it, along with the research questions and fundamental objectives that guided its development. It also details the approach we have adopted to address the identified challenges. Additionally, this chapter presents the structure of the document.

1.1. Motivation

Introductory programming courses present intrinsically different challenges compared to those faced by proficient programmers. For many students, these courses serve as their first exposure to programming concepts, which generally involve multiple layers of abstraction [**4**] making seemingly simple problems appear quite demanding for individuals with less experience.

As is widely understood in the academic community, dropout rates for courses of this nature are notoriously high when compared to other undergraduate courses [**19**].

One of the main barriers faced by students lies in the often daunting task of interpreting error messages generated by their code [**7**, **11**], sometimes seen as enigmatic, providing little to no help to the untrained. Furthermore, traditional debuggers seem intimidating to beginners, as they pose a significant skill barrier to students unfamiliar with standard debugging procedures. This is particularly true for the default debuggers that come with the most popular Integrated Development Environments (IDEs), which generally rely on breakpoints and a step-by-step approach.

Another hurdle arises from the confusion associated with tasks outside the sphere of coding, such as setting up the programming environment. This is yet another cryptic step standing in the way of students early in their learning path, recurrently resulting in configuration problems. Given the limited number of hours per week of direct contact between students and teachers, time wasted on solving these issues may prove significant.

Over the past decades, numerous initiatives have focused on creating pedagogical debuggers that prioritise usability and simplify complex concepts, usually delivered in the form of an external tool meant for being used as part of another IDE. The concept of a unified debugger and code editor is not novel in and of itself, as its the benefits have been pointed out before [**1**]. Nevertheless, as far as we are aware, such an approach has never been implemented within the context of an inspection-based debugger.

We believe that encouraging users to explore the execution details of their implemented methods can aid their development as programmers. Additionally, this system may also serve as a tool for teachers to better identify bugs in their students' code.

1.2. Research Questions

The primary goal of this dissertation is to investigate the feasibility of implementing a pedagogical programming environment centred on the notion of code inspections. We also wish to explore the impact of such features on the target user base, i.e., students at the introductory programming level. To this end, we aim to examine the following research questions:

RQ1 How could variable tracing be effectively implemented in a pedagogical environment?

RQ2 What impact do variable tracing features have on debugging tasks of introductory programming exercises?

1.3. Objectives and Approach

In the interest of tackling the issues presented above, we developed a debugging system that is fully integrated with a standalone environment for Java programming. Compared to traditional IDEs, one of the major advantages of using a standalone application is the lack of configuration steps regarding the environment itself (e.g., path variables, directory management, etc.).

We aimed to lift many of the usual troubles that come with debugging, such as the often inconvenient mode switch from the code editing area to a separate debugging view. By combining the two modes, the transition between writing code and fixing potential bugs is consistent and harmonious. Users should feel that the inspections are constantly available and ready to be toggled on from a single button click.

Regarding the debugging approach itself, the goal was to present a holistic behavioural analysis of the inspected methods, rather than focusing on user-selected locations (e.g., using breakpoints). The debugger was developed with the aim of serving as more than a tool for detecting the source of issues, rather acting as a mechanism to thoroughly understand every aspect of a program, such as tracing the history of variable assignments to evaluating an expression at any point in the code, and more.

In addition, the incorporation of a structure for writing unit tests was intended to encourage the adoption of Test-Driven Development (TDD) practices during the initial stages of learning. TDD has been shown to be effective in multiple contexts [**1**], although there are some concerns regarding its applicability in beginner practitioners [**3**]. We contend that our interface offers an accessible approach to this technique and, given the nature of the exercises posed to first-year students – generally simple arithmetic and logic problems – unit testing is particularly useful for validating the correctness of their solutions.

This project was developed as part of a broader Kotlin application, building upon an existing standalone coding environment. Although the application's framework was already established, our primary contributions included implementing the argument parsing logic

for handling input parameters within multiple contexts in the environment, the debugging system, and and the unit test structure.

1.4. Document Structure

Chapter 2 presents our literature review on programming education challenges, pedagogical debugging, and existing tools related to our work. Following, in Chapter 3, we give an overview of the user experience with the environment, describing its layout and core functionalities. Chapter 4 then explores key implementation details of the application, and the essential technologies used to develop its features. In Chapter 5, we cover a usability study conducted with programming instructors to assess the effectiveness of our proposed tool. Finally, Chapter 6 presents our conclusions and outlines directions for future work to enhance the prototype's capability.

CHAPTER 2

Background

In this chapter, we explore the background for the presented work, examining the existing literature on the challenges of learning and teaching programming, the topic of pedagogical debugging, and also include a section on a Kotlin library that was crucial to this work. Additionally, we present an overview of existing tools aimed at achieving similar goals to ours, highlighting the key characteristics of the reviewed debuggers.

2.1. Literature Review

It is broadly recognised that introductory programming courses present notable difficulties to students [**8**, **19**]. As pointed out by Jenkins (2002) [**8**], programming is not merely a single skill or a simple set of skills, but rather a hierarchy of skills. Considering this view, it is logical to approach teaching by gradually increasing the level of complexity and focusing on mastering the fundamentals in the initial learning stages.

Lahtinen et al. (2005) [**11**] analysed the results of a survey conducted on 559 students and 34 teachers regarding "difficulties experienced and perceived when learning and teaching programming". The questionnaire received responses from six universities, with the students having some programming experience either in Java or C++. First and foremost, the paper points out that the often large dimension and heterogeneity of student groups make it difficult to design beneficial instructions for everyone, recurrently leading to high dropout rates on courses of this nature. Regarding the survey's results, it was discovered that both the group of students and the group of teachers believed that finding bugs in a program was one of the "most difficult issues in programming".

One of the key skills of experienced programmers is the ability to mentally trace the history of specific variables in a program, or even the program in its entirety. Beginners' tracing skills have been shown to be often quite poor, as it is an ability that develops with experience. Vainio et al. (2007) [**18**] attempted to find out the reasons for this adversity. The article defines tracing as "mental execution", as in, a simulation of the program's execution steps. The study found that some students struggled with non-trivial variable assignments, i.e., "those that are results of some computation and whose values cannot be seen in the program text". In the tool presented in this document, we address these situations by allowing users to see the values being assigned to variables and tracing their history throughout the execution.

The concept of variable roles has also impacted our research from the beginning. Identifying the purpose of each variable has been repeatedly shown to improve students' mental grasp of the concepts they are confronted with in programming exercises [**2**, **9**].

Certain program behaviours can be represented by roles that illustrate variable functions that are widely found in academic exercises. During the development of this prototype, we were mindful of the potential advantages of incorporating variable roles into the debugging experience.

There have been numerous success stories in using visualisation tools for introductory programming courses. Yadin (2011) [**19**] described a study which resulted in a reduction of the percentage of failing students by 77.4% in an introductory course, over a period of four years. Even after shifting from Java to Python, focusing more on algorithms and imperative programming as opposed to teaching in the light of the object-oriented paradigm, it was not until the students started using the GvR visualisation tool[1] that the failing rates showed a dramatic drop.

Likewise, a study conducted in our facilities by Santos (2011) [**16**] evidenced the benefits of the usage of the Aguia/J tool for Java. To attend the final exam in this course, students must succeed in the lab class evaluation. Pilot groups using the tool reached the exam in larger proportions than the control groups, as well as being more successful in the exam itself.

2.2. The Strudel Library

The backbone of this project lies in the Strudel library[2] for Kotlin, which comprises an architecture for modelling structured programming – a programming paradigm based on the idea that a program should have a hierarchical structure and extensively use control flow mechanisms (e.g., `if/else`), loops (`for/while`), and reusable modules –, allowing for precise observation of execution events of a programming model.

To employ these functionalities, one can load a Java source code file into an instance of a Strudel virtual machine, retrieving a module from which a procedure may be executed. Figure 2.1 is a code snippet that illustrates this mechanism.

Several aspects of the execution can then be examined, particularly by tracking variables, loop iterations and error messages. As will be expanded on in section 4.3, in the context of our project, we collected this information and handled it accordingly to present the code inspections.

Another major feature of this library is its ability to detect variable roles. Although we do not explicitly refer to these roles as done in the reviewed literature, the code inspections are presented with slight differences that highlight the behaviour of the variables (e.g., iterators, accumulators, etc.). Such differences are explained in section 3.4.1.

Strudel supports a limited set of Java instructions, covering the basic keywords and data structures generally recommended to be taught to students at an early stage. Consequently, there are some constraints on how a program can be developed using our tool. However, we believe these limitations are justified, given that it was designed to be particularly useful to beginners, by providing mechanisms to help understand the fundamentals of

[1] `https://gvr.sourceforge.net/`
[2] `https://github.com/andre-santos-pt/strudel`

```
// Java file to be loaded
val file = File("Test.java")

// Loading the Java file into a Strudel module
val module = Java2Strudel().load(file)

// Retrieving the method (procedure) to be executed
val method = module.getProcedure("foo")

// Creating the Strudel virtual machine
val vm = IVirtualMachine.create()

// Execute the method
val result = vm.execute(method)
```

FIGURE 2.1. Code snippet illustrating how a Java procedure can be executed using the Strudel library.

programming. It is meant to serve as a controlled environment where novice programmers can build confidence in their knowledge and skills before progressing to conventional, less beginner-friendly IDEs.

2.3. Visualisation Tools

This project builds on previous efforts in the development of pedagogical program visualisation tools. In this section, we review some of the most notable resources in this realm, focusing on specific aspects that we sought to explore in our project.

It is relevant to note that this prototype was developed with the prospect of being a potential successor in the line of programming tools that have been used by first-year students at our facilities.

2.3.1. PandionJ

PandionJ [**17**] was, in many ways, a precursor to the tool we present in this document. First and foremost, it has been used at our facilities by students enrolled in the introductory programming course for several years. Its use highlighted several areas of improvement, inspiring us to develop an even more comprehensive and user-friendly environment. Many of the functionalities we explored were first implemented in PandionJ, most notably the focus on variable roles to visually display variables illustrating their characteristics.

It was built as an Eclipse[3] plugin that integrates with its debugger infrastructure. This is one of the main differences between PandionJ and our tool, as we wanted to distance ourselves from a third-party IDE, for the reasons mentioned in the above chapters. PandionJ works together with the breakpoint debugging feature in Eclipse, providing information about variables' history and a look-ahead into their future state. This information is inferred through static analysis of the source code, deriving relationships between code elements and their behaviour.

[3] `https://www.eclipse.org/`

The debugging process is achieved via a view consisting of graphical renders of program variables. At a given suspended state of the program, users can trace the current value of a variable and, depending on their role, gather information on their past and future values.

Figure 2.2 is a screenshot of the PandionJ tool which exemplifies its tracing features, representing variables differently according to their role. For instance, "sum" is recognized as an accumulating variable (gatherer), illustrated by the sum expression in parenthesis "$(0 + 3 + 5)$".

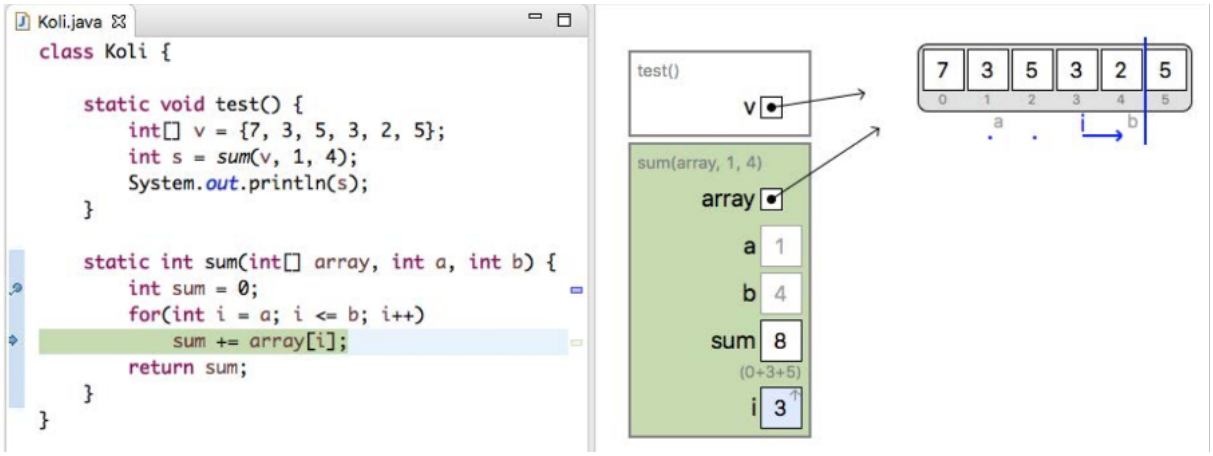


FIGURE 2.2. A screenshot of the PandionJ tool illustrating a method for summing the values contained within a given interval of array indices. Figure 1 in [**17**].

2.3.2. Aguia/J

Before PandionJ, students at our facilities used Aguia/J [**16**] in the introductory course, which is an experimentation and visualisation tool tailored towards learning object-oriented concepts in Java.

The debugging approach of Aguia/J consists of an interactive view where users can experiment with creating objects and updating their properties, without having to stop the execution (Figure 2.3).

Users can see the effects of making changes to the source code via a visual representation of the declared classes and objects. Cues such as greyed-out boxes for private attributes and checkboxes for booleans constitute illustrations of programming concepts and complement the pedagogical experience.

2.3.3. TOD

TOD (Trace-Oriented Debugger) [**13**] is an Eclipse plugin that allows users to navigate through the execution of a program. It is described as an "omniscient debugger", in the sense that the tool records the entire execution trace of a program and lets users freely explore it, without the need to re-run it multiple times to pinpoint the source of a bug. While the implementations differ abundantly, this philosophy closely aligns with our approach.

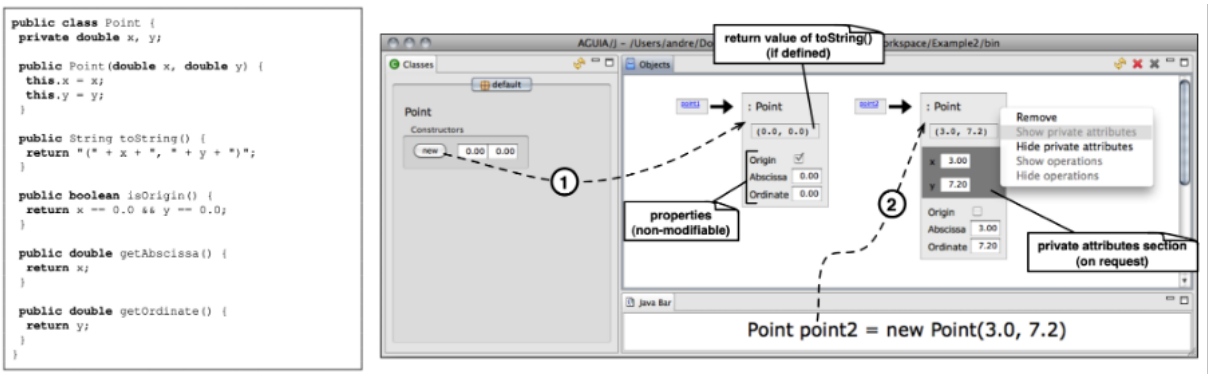


FIGURE 2.3. A screenshot of the Aguia/J tool. Java source code (on the left) is visually represented in the GUI (on the right). Figure 2 in [**16**].

TOD provides a feature to jump directly to events that assign current variable values, allowing users to look-back into the causes of faulty behaviour. Users can also bookmark events and objects to access them quickly, which can be useful in large programs.

Apart from these standout features, the debugging process follows the traditional approach of stepping backward and forward in time through "web browser-like buttons". Figure 2.4 exemplifies such a navigation.

2.3.4. Memview

Memview [**6**] is a visual debugging tool, created as an extension to the DrJava [**14**] IDE[4]. Essentially, it is an interactive display of computer memory, divided into three panes: one for the call stack, another for static objects allocated in the heap and a last one for regular heap objects. Figure 2.5 is a screenshot of the tool in action.

The debugger is meant to help students understand Java concepts, especially object-oriented programming. It aims to teach students key ideas, such as memory addresses, references, the heap, etc.

Although the tool's focus significantly differs from ours, it is worth mentioning that a user experiment conducted in an introductory programming course using it yielded notable success. Empirical analysis suggested that the tool was indeed able to provide a helpful visual depiction of the memory model, resulting in a better understanding of the concepts at play.

2.3.5. JIVE

JIVE (Java Interactive Visualization Environment) [**5**] is a pedagogical tool designed to aid the understanding of Java programs through dynamic visualisation, presenting visual representations of object structures and their states, as well as the call history of a program via automatically generated diagrams, as can be seen in Figure 2.6[5].

Contrasting with traditional step-by-step debugging approaches, JIVE offers a declarative method based on a set of queries over the program's history. For instance, users can

[4] `https://drjava.sourceforge.net/`
[5] Retrieved from `https://cse.buffalo.edu/jive/`.

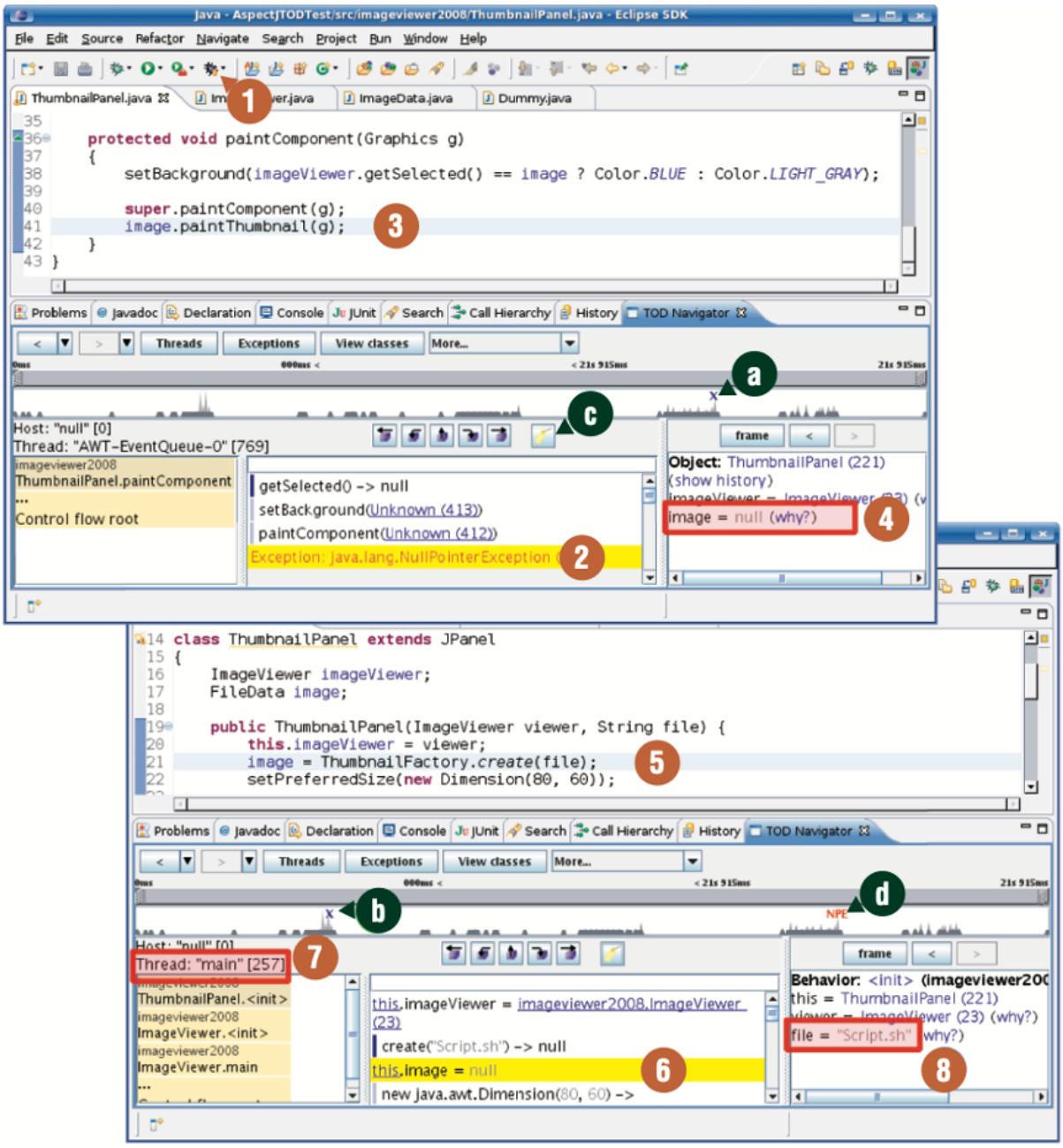


FIGURE 2.4. Navigation history in TOD. Extracted Figure 3.a from Pothier and Tanter (2009) [**13**].

utilise these queries to obtain information about an object at a given execution state or retrieve all values assigned to a variable throughout its lifetime.

Though powerful, JIVE cannot be considered a beginner-friendly tool, as most of its features require a certain level of experience and an understanding of the framework itself.

2.3.6. BlueJ

BlueJ [**10**] is an educational IDE intended for teaching object-oriented programming in Java. It is built upon a standard Java Software Development Kit (SDK), thus using a

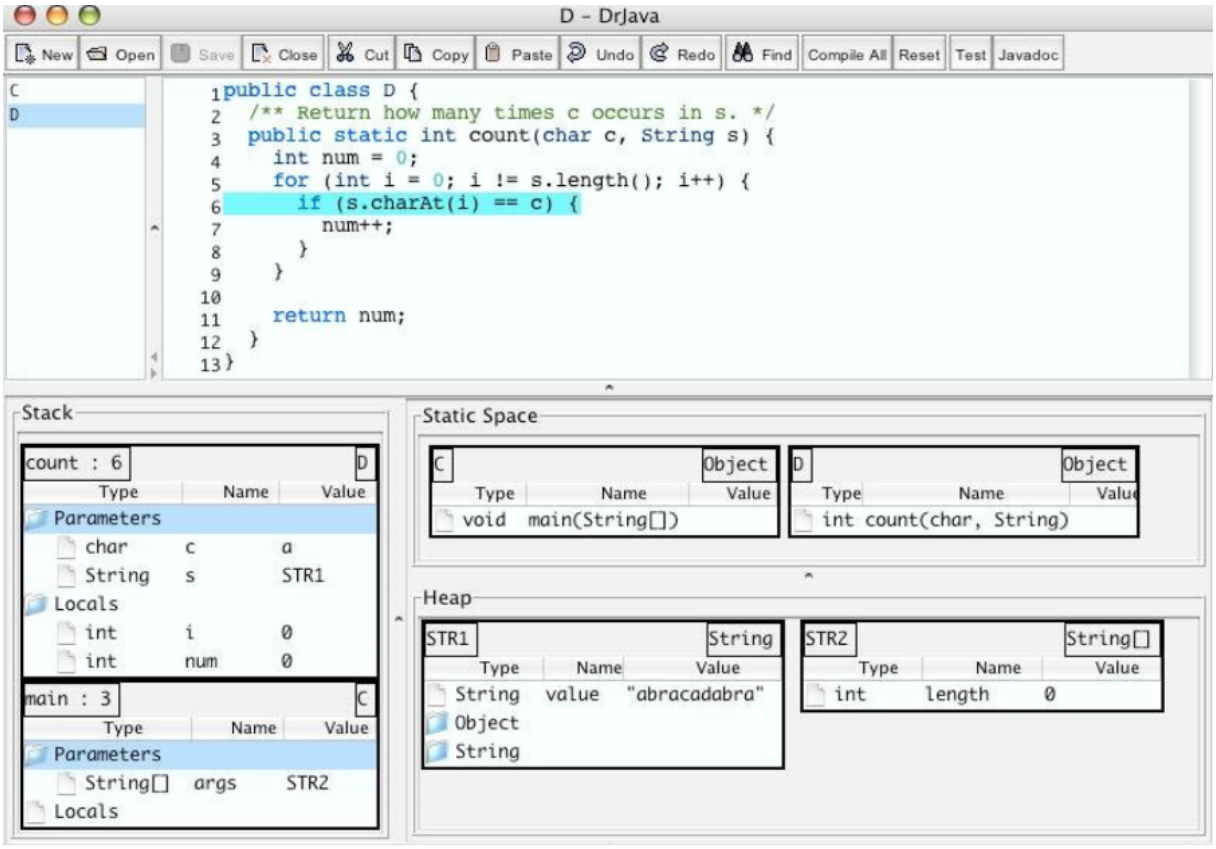


FIGURE 2.5. Debugging a program with Memview. Extracted Figure 3 from Gries et al. (2006) [**6**].

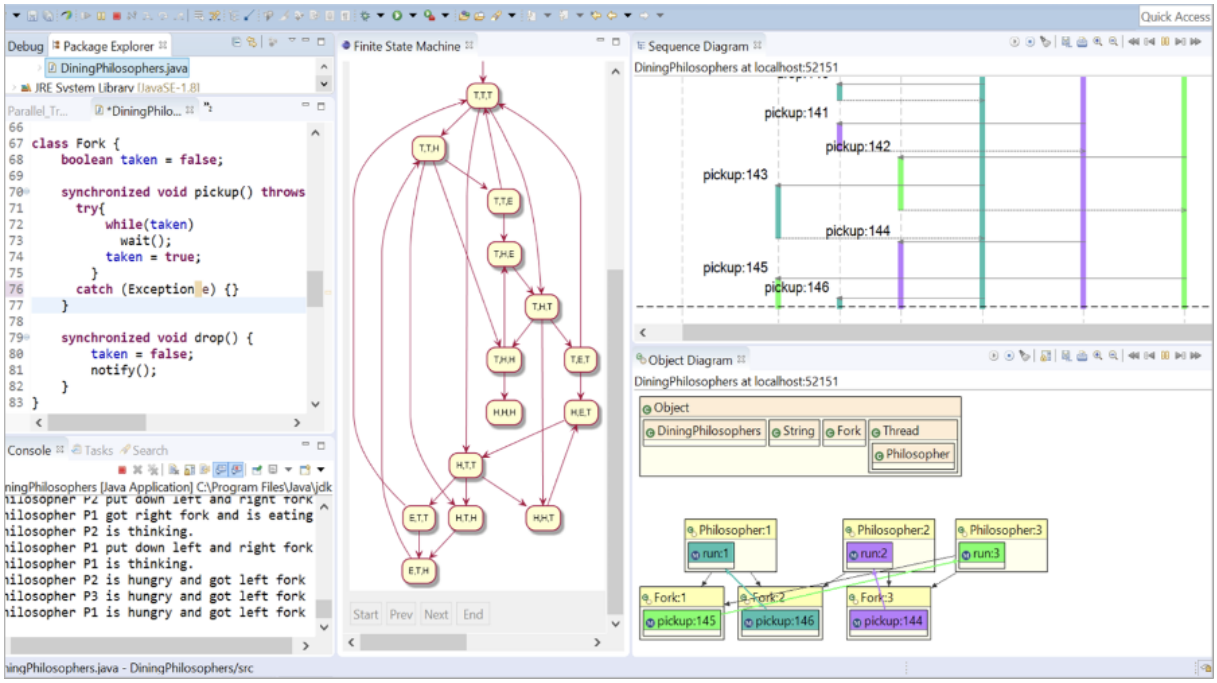


FIGURE 2.6. Interactive visualization in JIVE.

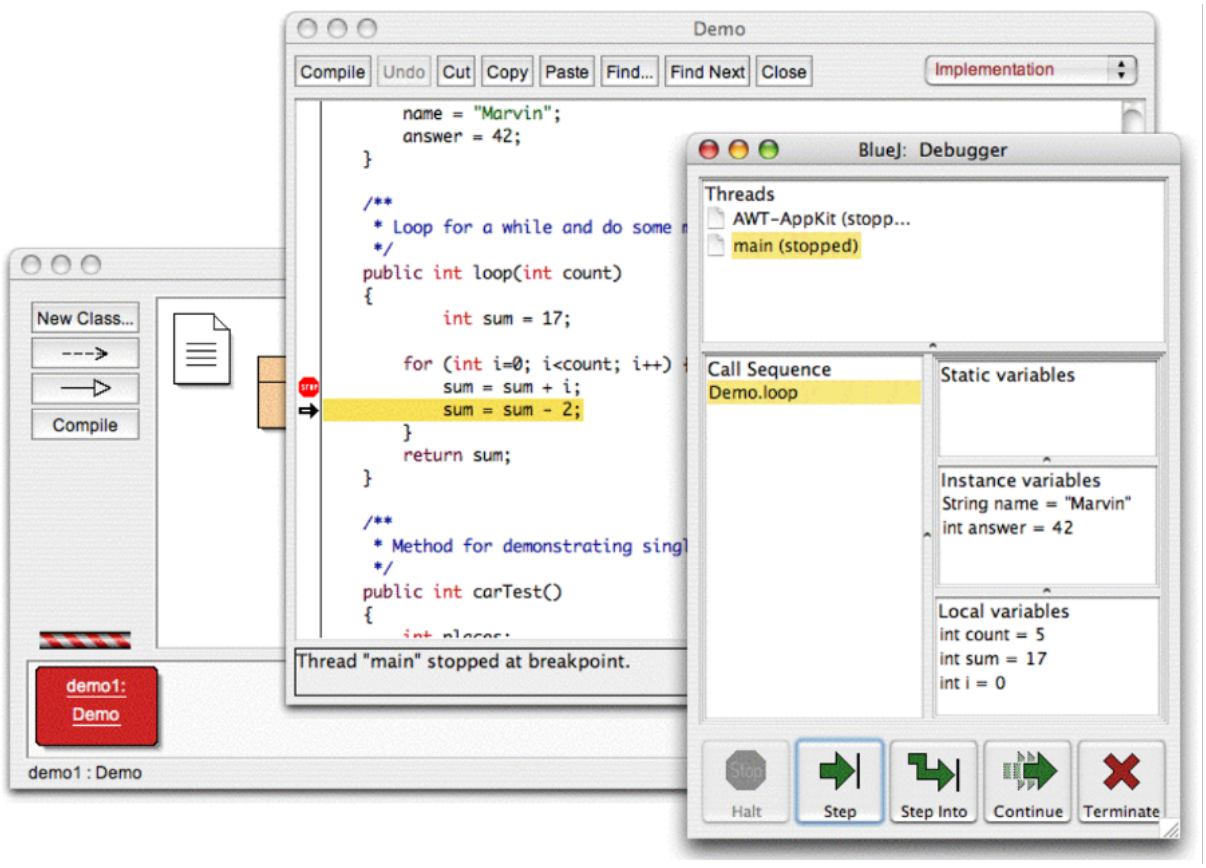


FIGURE 2.7. Interactive visualization in BlueJ.

standard compiler and virtual machine, but presenting a custom User Interface (UI) which offers visualisation features.

The approach is based on a window containing a UML class diagram of the project structure. From this diagram, users can instantiate objects and execute any public methods from that class. An object's properties can be inspected at any time through the object's, including its static and instance fields.

BlueJ also features a built-in debugger which supports step-by-step execution and breakpoint-based debugging, as shown in Figure 2.7[6]. The seamless integration of the debugger in the IDE encourages students to focus on learning without being overwhelmed by complex tools.

2.3.7. Jeliot 3

Jeliot 3 [**12**] is a program visualisation and animation tool tailored to Java and designed to aid beginner students to learn procedural and object-oriented programming. The key idea is to engage students in building their own programs while also examining a visual representation of the program's execution, helping them develop a mental model of programming concepts and associating them with concrete implementations.

[6]Retrieved from `https://teaching.csse.uwa.edu.au/units/CITS1001/handouts/BlueJDebuggerTutorial.pdf`,

The tool has evolved from previous versions, Jeliot I and Jeliot 2000, focusing increasingly more on novices and fixing shortcomings found in empirical evaluations of these installments. Jeliot 3 extends the features of Jeliot 2000 to allow visualisation of object-oriented concepts.

Figure 2.8[7] is a screenshot of the user interface of Jeliot 3. The visualisation area is organised in four sections, namely a method frame area, an expression evaluation area, a constants area, and an instance area.

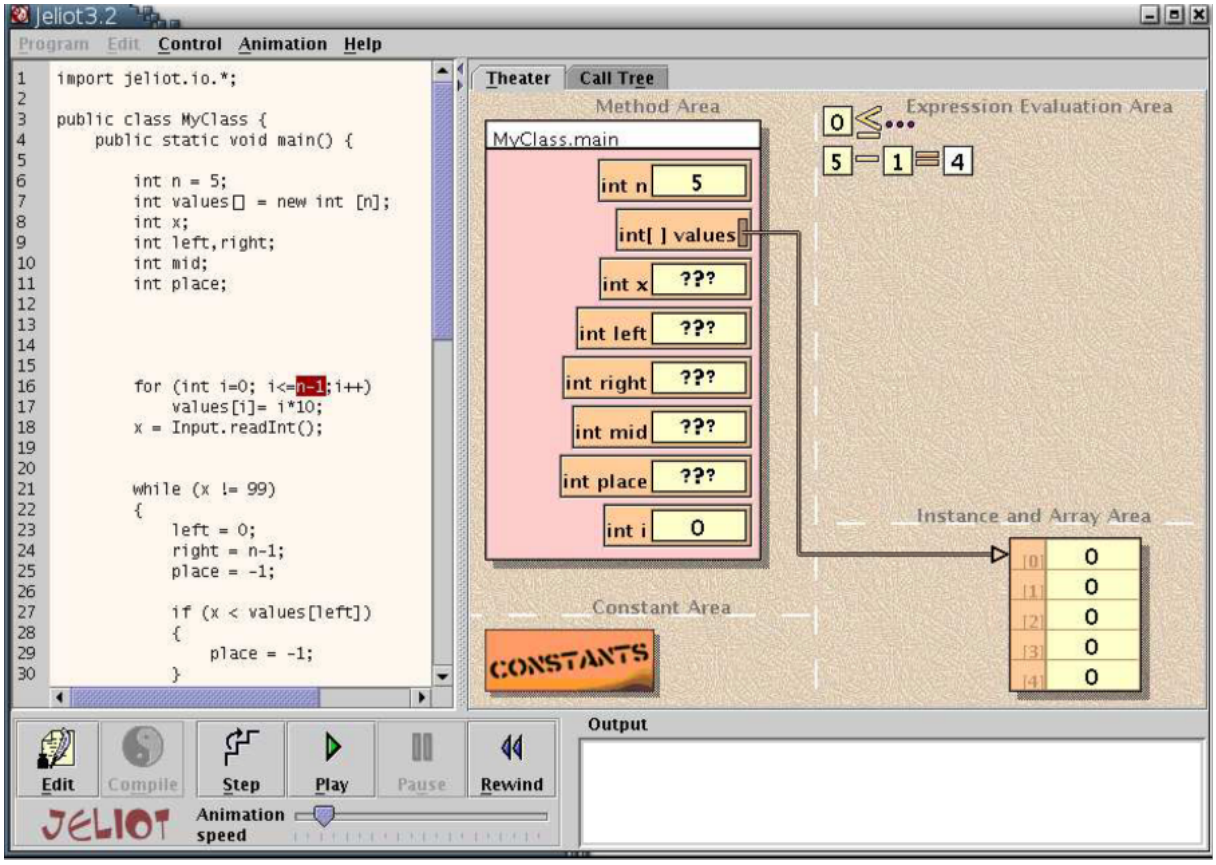


FIGURE 2.8. Jeliot 3 GUI.

2.3.8. PlanAni

PlanAni [**15**] is a program visualisation system based on variable roles. It creates animations tailored to the roles of the variables being illustrated (Figure 2.9). For instance, stepper variables are depicted by footprints showing its current, previous and potential future states, along with an arrow pointing in the direction to which the values are evolving. Likewise, fixed-value variables are represented by a stone, conveying the idea that it will not change during the execution.

An experiment was conducted with three groups: one group was instructed using traditional methods, another using variable roles as a concept to complement the course, and a final one using variable roles along with the animator. The teachers found that

[7]Retrieved from `https://epl.di.uminho.pt/~gepl/GEPL_DS/PEP/teste/JeliotPEP/userguide.pdf`.

the visualisation system made programming concepts easier to understand and entailed livelier discussions. PlanAni users demonstrated a stronger grasp of the program's overall behaviour and the contribution of each variable.

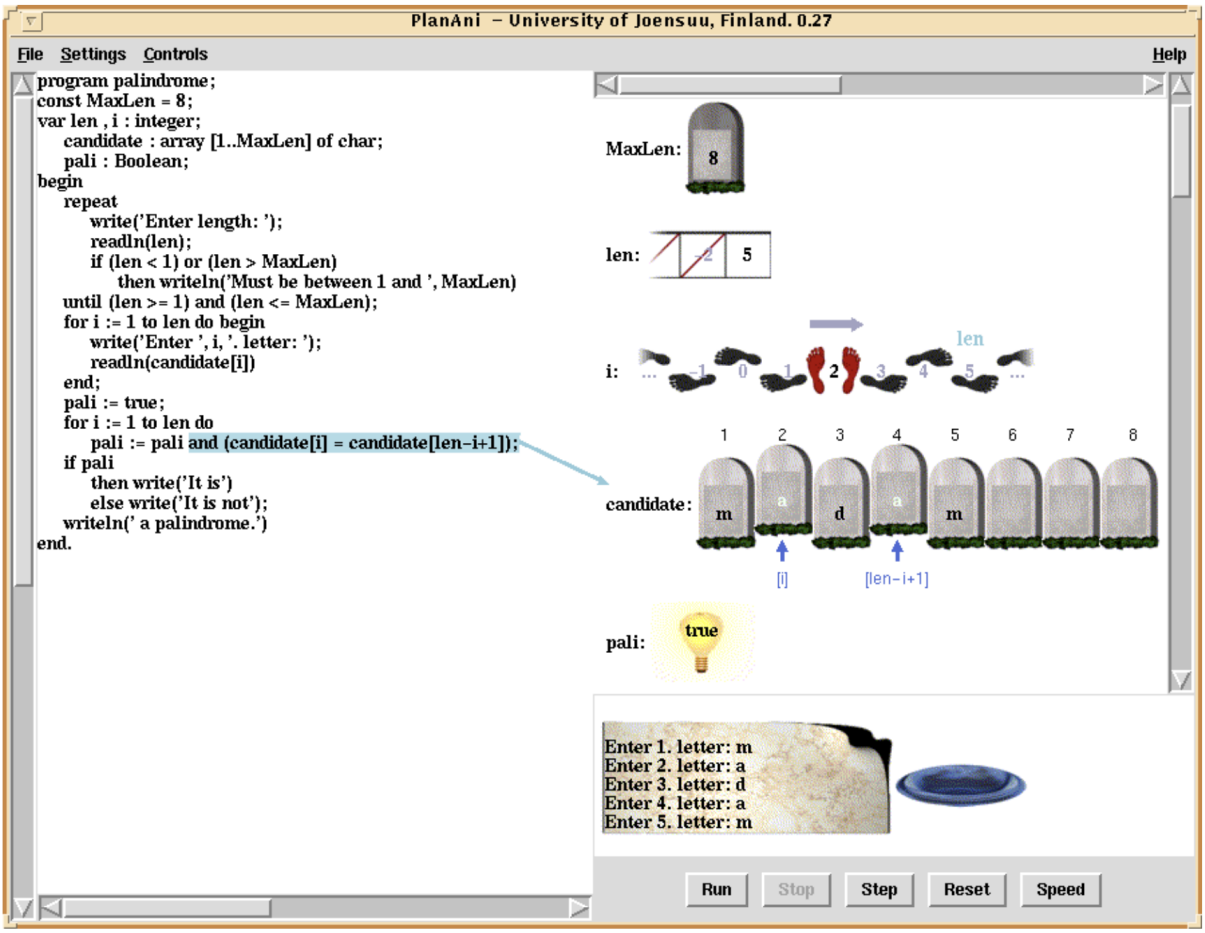


FIGURE 2.9. PlanAni GUI.

2.3.9. Overview

The previous subsections reveal the existence of various tools that have been developed with a focus on pedagogical visualisation features. Multiple approaches have been tested in view of fostering the learning path of beginner programmers, ranging from debuggers to diagram generators and more.

Our key takeaways from researching this topic are that a great portion of such tools rely on external IDEs, thus requiring configuration before they can be utilised. Furthermore, many traditional approaches may not be as beginner friendly as their description might suggest. One of the main aspects we noticed when analysing these tools is the often crowded, confusing, and archaic-looking user interfaces, which undoubtedly hinders the experience users have with them. In designing our prototype, we focused on addressing these issues by simplifying the user interface, aiming to reduce the entry barrier and offer a cleaner, more intuitive experience.

Moreover, we noted the success cases of using variable roles as a means to complement the learning process of programming concepts. By incorporating variable roles within our prototype, we aimed to create an environment where students could more easily identify the function of each variable throughout a program, particularly those exhibiting behaviours that recurrently occur across different programs.

CHAPTER 3

The Environment

This chapter explores **RQ1**, providing an overview of the user experience with the developed environment and explaining the content layout of the application and its core functionalities, such as running methods, unit testing and code inspection.

3.1. Layout

The application's layout consists of a code editor on the left, a table for writing unit tests on the right and a toolbar at the top (Figure 3.1). The toolbar includes, in order, a button for opening a Java file, followed by buttons for running a method, opening the settings menu, and toggling code inspections. The following sections cover the main user experience aspects of every notable feature of the prototype.

3.2. Running a Method

To run a method, users should place their cursor inside the method's body and then press the green "play" button in the toolbar.

Depending on if the selected method takes parameters, a window will pop up, prompting users to write the method's arguments, as seen in Figure 3.1, where the method `sequence` was invoked. If the introduced arguments are valid, the method will be executed upon pressing "Submit". Otherwise, the text boxes where the invalid arguments were passed will

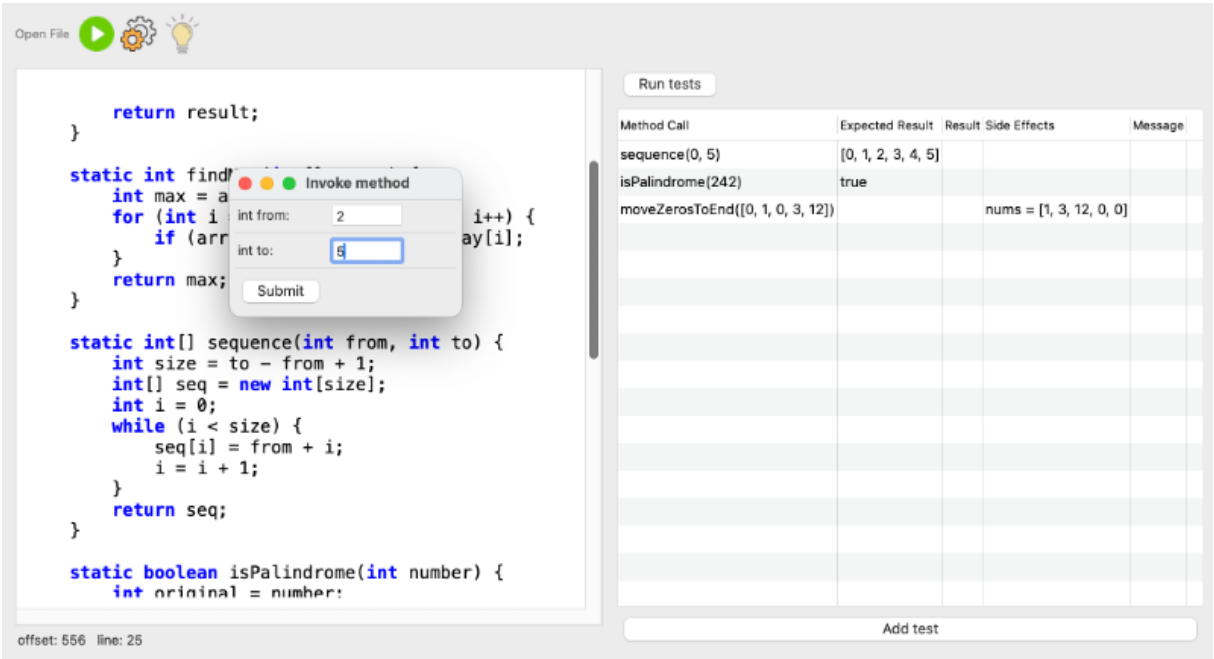


FIGURE 3.1. Layout of the application.

be marked in red and the method will not execute. For methods that take no parameters, this step is skipped.

The application supports four primitive types – `integer`, `double`, `char` and `boolean` – as well as one-dimensional arrays of each mentioned type.

In the interest of simplifying the user experience, we opted not to follow the Java syntax for declaring arrays, which involves using the `new` keyword and explicitly stating the type of the array (e.g., `new int[] 1, 2, 3`, in the case of an integer array). Instead, in the application, array arguments should be passed as shown in Figure 3.2.

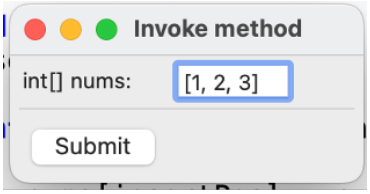


FIGURE 3.2. Syntax for passing array arguments.

After the method executes, the result will be shown on top of its declaration via a tooltip (Figure 3.3). Users may then wish to inspect the method's behaviour, which can be done by pressing the inspections button - represented by a yellow lamp - which should now be enabled. Being one of the key features of this prototype, the code inspections will be thoroughly explained in section 3.4.

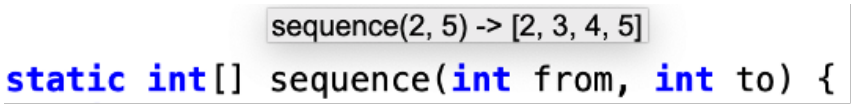


FIGURE 3.3. Visualisation of method execution results.

3.3. Unit Test Table

As previously mentioned, we aimed for our application to encourage early-stage learners to experiment with test-driven development, leading to the inclusion of a unit test table.

3.3.1. Table Layout

The table, shown on the right side in Figure 3.1, consists of five columns – "Method Call", for typing the unit test; "Expected Result", for inputting the expected return value of the method; "Result", for showing the actual return value; "Side effects", for inputting the expected final value of a method's variable(s); and "Message", for displaying potential execution error messages. The "Result" and "Message" columns are read-only, while the others may be edited.

Additionally, there are two buttons that interact with the table — "Run tests", on top, for running the test battery; and "Add test", below, which allows the next line on the table to be edited.

3.3.2. Running Tests

Figure 3.4 demonstrates a use case for the test table. In this example, the method `findMax` is intended to find the maximum value in a given integer array, and `factorial` should calculate the factorial of an integer input. The table contains three tests for each method. As indicated by the colour of the values in the "Result" column — green for correct and red for incorrect – the final test, `factorial(4)`, failed, as the method did not return the expected result. This suggests the presence of a bug which lead it to compute an incorrect value. Furthermore, the first call of the `factorial` method, in line 4, has a typing error, triggering the display of an error message.

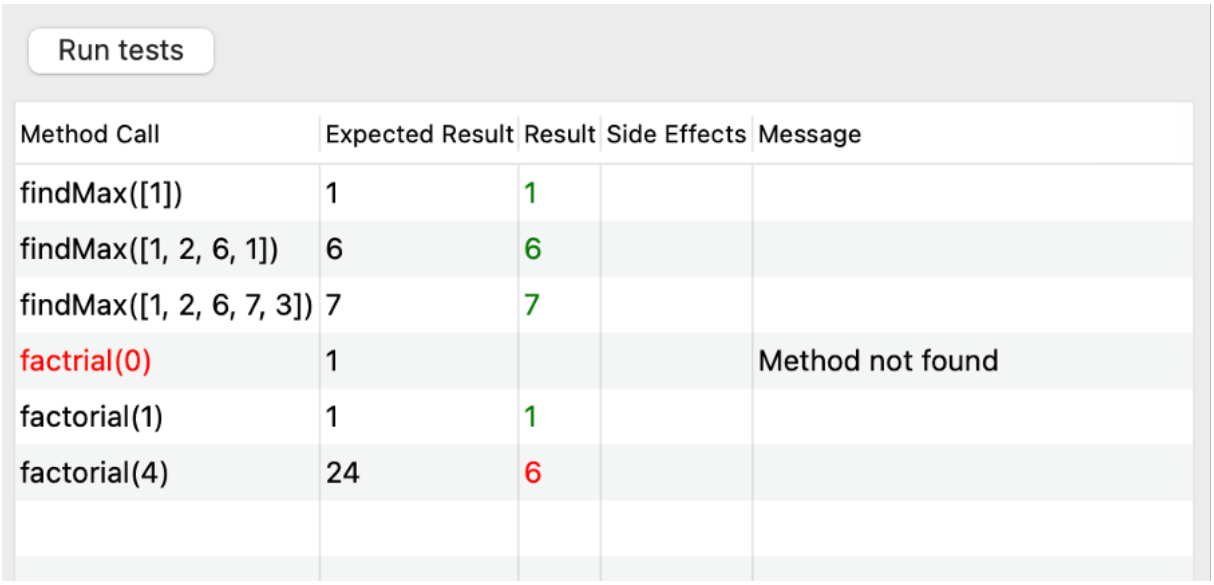


FIGURE 3.4. Usage example of the test table for the methods `findMax` and `factorial`.

To investigate further, users can select the faulty execution by clicking on its corresponding table row and pressing the inspection button in the application's toolbar, triggering code inspections to appear in the editor. Users may also examine other executions, whether unsuccessful or not, by selecting different rows in the table, alternating between various test results, and allowing them to analyse each one individually.

This process illustrates the seamless integration between the editor, the unit test table and the code inspections.

3.3.3. Side Effects

Unit tests are typically targeted towards methods that return some value. However, void methods could also be tested by examining the impact on other variables inside of the method, rather than the return value. We included a "Side Effects" column that allows users to evaluate the expected values of any variable after the method's execution.

To test for side effects, users should input tests using the following notation: `varName1 = value1; varName2 = value2; ... varNameN = valueN`. As long as the variables exist within the method, they can be tested in the same way the return value can.

Figure 3.5 exemplifies this feature in the case of a sorting algorithm ("bubble sort"), where the final value of the input array parameter (`int[] arr`) is checked, despite the method having a `void` return type. The cell's text is displayed in green when the test is successful, or red when it is not.

FIGURE 3.5. Usage example of the "Side Effects" column.

3.3.4. Error Messages

Bearing in mind the subjacent philosophy of this project of simplifying otherwise complicated programming activities, we aimed to display helpful error messages wherever justified, and that also applies to the context of the unit test table.

Parsing errors often result from users mistyping a method call or a parameter value, and the problem may not always be clear for a novice. Instead of merely stating that there is a problem, we focused on providing personalised explanations on why something might have gone wrong. Users may find the following error messages in the "Message" column.

- "Invalid method call";
- "Method not found";
- "Number of arguments do not match number of method parameters";
- "Invalid argument type(s)";
- "Invalid side effect";
- "Side effect variable not found".

3.4. Code Inspections

In a Java method, numerous aspects can be inspected, each requiring different approaches according to their characteristics, in order to provide the most helpful analysis to users. We explore these particularities by experimenting with how such concepts can be illustrated in a suggestive manner, namely variable and expression tracing as well as runtime error handling.

As previously stated, in a Strudel model every variable is assigned a role based on its behaviour throughout the program's execution. Table 3.1 explains every contemplated variable role within the scope of this application.

In the context of this section, the term "literal" is used in relation to an expression written explicitly in the code, rather than being computed from other values (e.g., `5` is a literal expression, whereas `sum + i` is not). This distinction is relevant for deciding where to display inspections.

To explore this feature, users should press the inspection button after either executing a single method directly from the code editor or running a battery of tests from the test

TABLE 3.1. Variable roles and their descriptions.

Role	Description
Fixed Value	Holds a constant value.
Stepper	Value history is sequential.
Gatherer	Holds a value obtained by accumulating several values.
Most Wanted Holder	Holds the most desirable value encountered so far during a sequence of comparisons.
None	Does not fit into any of the above descriptions.

table and then selecting the execution to be analysed. Only one method can be inspected at a given time.

The following subsections discuss every inspection-worthy situation covered by the debugger and explain the decisions behind certain design features.

3.4.1. Variable Tracing

Apart from when a fixed value (constant) variable is assigned a literal expression – in which case, no further inspection is needed, as the variable's behaviour is trivial – users should expect to find code inspections in the form of tooltips that reveal the history of every variable in a program. These tooltips are revealed by clicking on the name of the variable to be inspected, within the statement where it was declared, and are shown directly beneath it.

Although fixed value variables are, by definition, assigned only once (and, therefore, their value history consists of a single value), when assigned a non-literal value, as illustrated in Figure 3.6, the debugger also sets a tooltip showing the actual value being assigned. Otherwise, no tooltip is shown, as its content would be redundant.

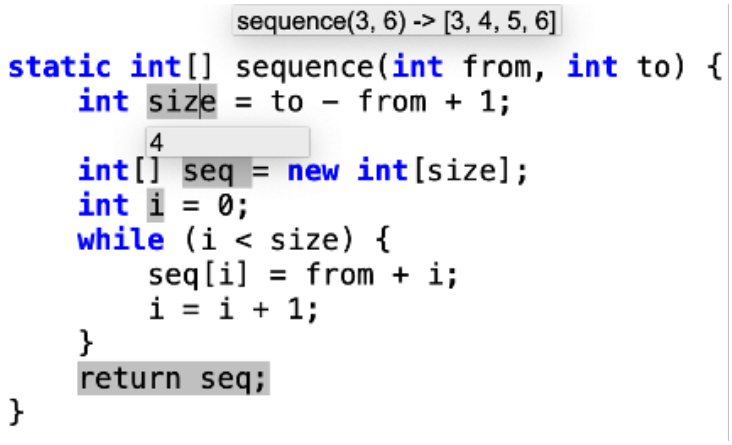


FIGURE 3.6. Inspection of a constant variable. `int size` is a constant, however its assignment value is not literal.

In the context of non-constant variables, their tracing history is also presented according to their role. For variables with no specific roles (see role "None" in Table 3.1), their value

history is displayed as shown in Figure 3.7. Every other case is explained in the next subsection.

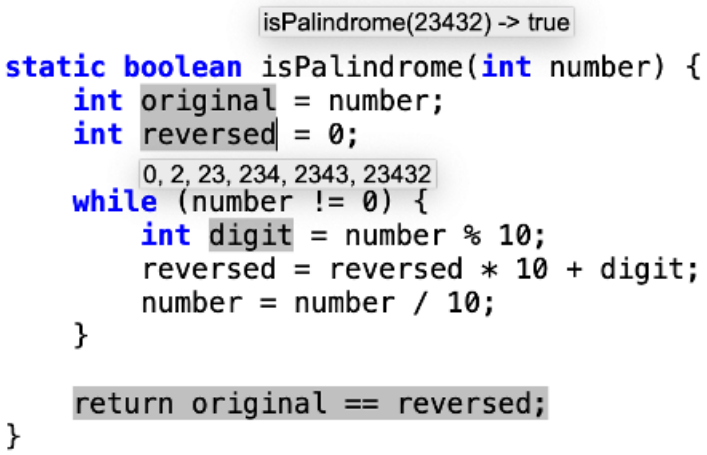


FIGURE 3.7. Tracing the history of values of a variable with no specific role – `int reversed`.

It is worth noting that, in the interest of maintaining the usability of the tooltips, we set a limit of 20 values shown for variables with an extensive history. We determined that this would provide a comprehensive enough representation of the behaviour of these variables.

Regarding role specific tracing, we reasoned that variable roles evoke distinct tracing representations to aid in understanding the program as a whole. The different tooltip styles were designed to visually convey a variable's behaviour without explicitly labelling its specific role. **Stepper** variables are most commonly used in loops, where they function as iterators. As illustrated by Figure 3.8, the tooltip associated with these variables reveals their starting and ending values, separated by two dots (".."), followed by their step size when it is larger than 1. **Gatherer** variables are stem from an accumulation expression, and for that reason, their tooltips reveal the explicitly typed-out expression that originated the final value. Figure 3.9 exemplifies this case with a variable that accumulates the calculation of a factorial number. **Most Wanted Holder** variables obtain their final value after a sequence of comparisons, landing on the most desirable encountered value. This behaviour can occur in problems such as finding the largest element in an array. Therefore, for tracing variables of this nature, the tooltips are represented by a strikethrough list of values up until the last one, which is written in plain text, illustrating the process of elimination preceding the arrival at the final value. Figure 3.10 exemplifies this tooltip style.

To further assist users in comprehending the behaviour of a program, the tracing of variables within a loop can be performed collectively in a table format. Although this information is available in the individual tooltips of each variable, it can be more intuitive for some users to examine the history of these variables side by side, each row corresponding to a loop iteration. The table (shown in Figure 3.11) includes all variables

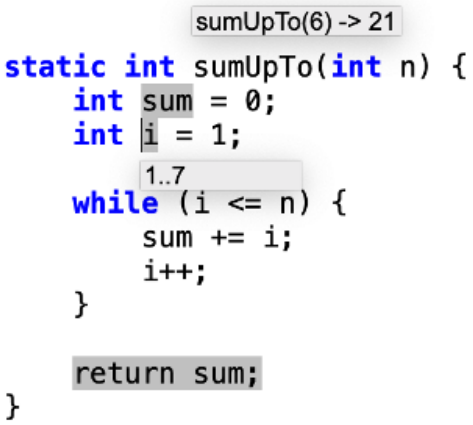


FIGURE 3.8. Tracing the value history of a stepper variable – `int i`.

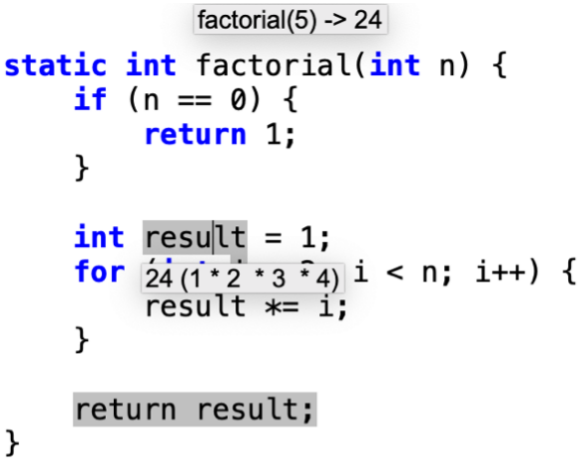


FIGURE 3.9. Tracing the value history of a gatherer variable – `int result`.

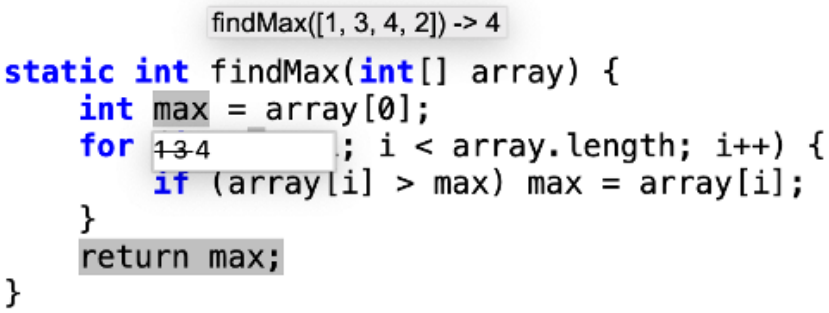


FIGURE 3.10. Tracing the value history of a most-wanted-holder variable – `int max`.

within the loop's scope. Each row displays their values immediately before the iteration begins.

Users can access this tooltip by clicking on the loop keyword - "for" or "while", depending on the chosen loop structure. Nested loops can also be inspected in the same manner, as illustrated by Figure 3.12.

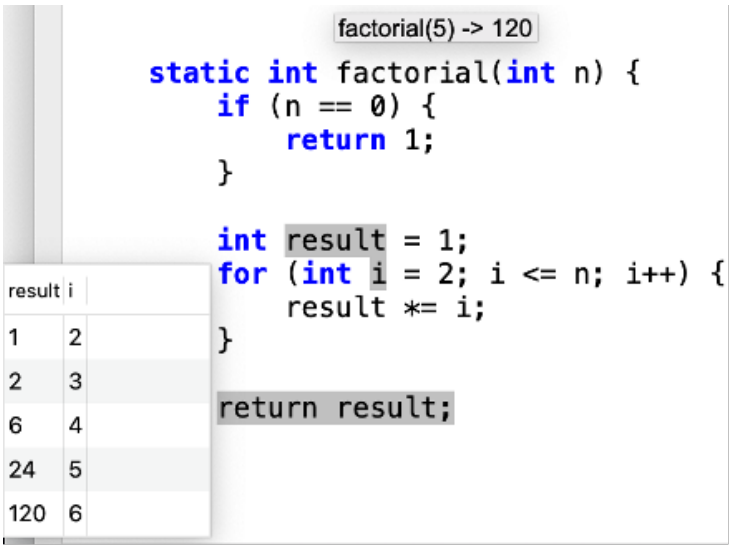


FIGURE 3.11. Inspecting a loop table.

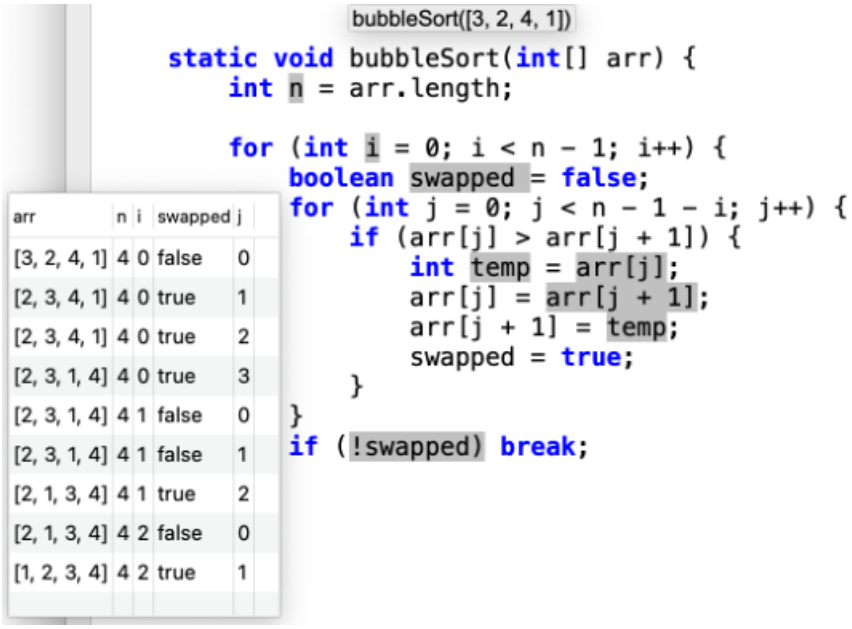


FIGURE 3.12. Inspecting a nested loop table.

Similar to the limit on the number of values shown in a regular variable trace, we set a limit of 20 lines for the loop tables, in addition to the line corresponding to the last loop iteration.

3.4.2. Expression Evaluation

Similar to the variable tracing feature, users may also access tooltips that display the history of expression evaluations. By clicking on any non-literal expression, they can inspect variable assignments, comparisons, and return statements.

Referring back to the example in Figure 3.6, the assignment expression may be inspected, revealing the explicit calculation being performed to reach the value to be assigned. Figure 3.13 portrays this tooltip.

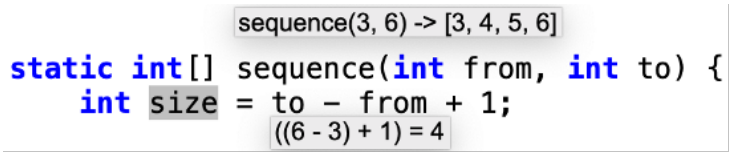


FIGURE 3.13. Multiple part assignment expression – `to - from + 1`.

For variable assignment expressions of this nature that occur inside loops, the tooltip reveals their evaluation history (Figure 3.14), showing how each part of the expression evolved throughout the loop iterations.

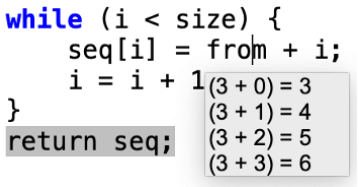
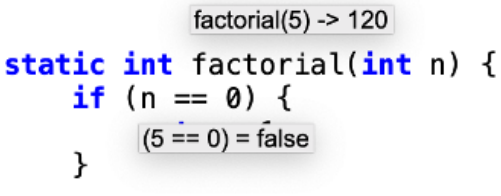
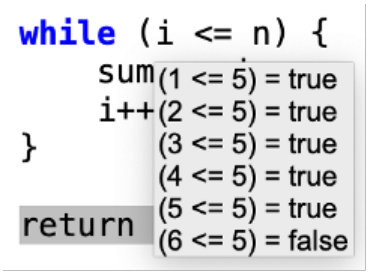


FIGURE 3.14. History of the assignment expression evaluation for `seq[i]`.

Likewise, conditions may also be inspected either outside or inside loops (Figure 3.15).



(A) Condition outside loop.



(B) Condition inside loop.

FIGURE 3.15. Inspecting conditions.

Finally, non-literal return expressions contain tooltips that vary depending on whether the expression is composed of multiple parts or just one (Figure 3.16).

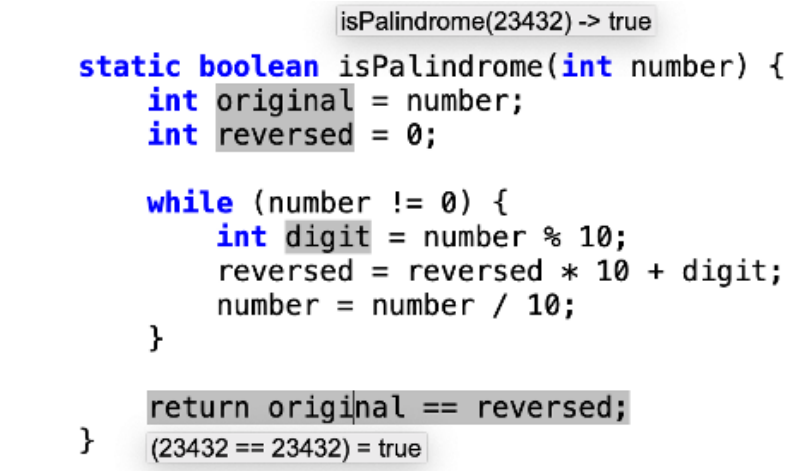
Expression histories are also limited to show only the first 20 evaluations, guaranteeing that the user space is never clustered with large inspection blocks.

3.4.3. Errors

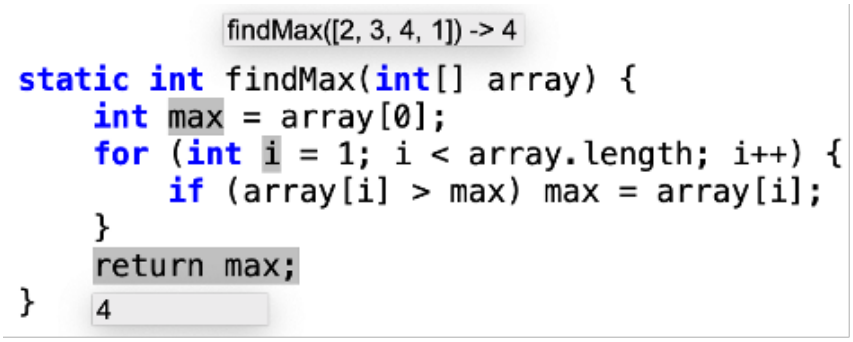
Whenever a runtime error occurs, the faulty section of the code is highlighted in the editor, accompanied by a tooltip explaining the error. This includes null pointer exceptions, array accessing issues, etc.

The images below illustrate some examples of such tooltips. Figure 3.17 demonstrates the error message shown when creating an array of negative size, and Figure 3.18 exhibits a situation in which an array is being accessed through an out of bounds index.

This approach is arguably more intuitive and beginner friendly than the traditional console-based approaches, where the location of the code that caused the error is shown in text form, as opposed to being directly displayed in the editor.



(A) Multiple part return expression.



(B) Simple return expression.

FIGURE 3.16. Inspecting return expressions.

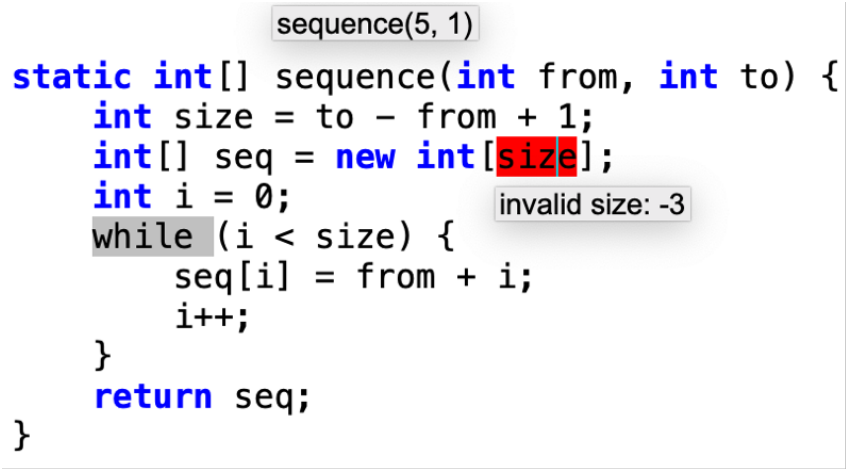


FIGURE 3.17. Error message obtained from attempting to create an array of negative size.

The features presented in this chapter capture the core objectives we set for the development of our prototype. Using this application, users are able to thoroughly inspect the behaviour of their code, enabling them to understand each action performed within a method, which not only supports the debugging task but can also enhance comprehension of the concepts involved.

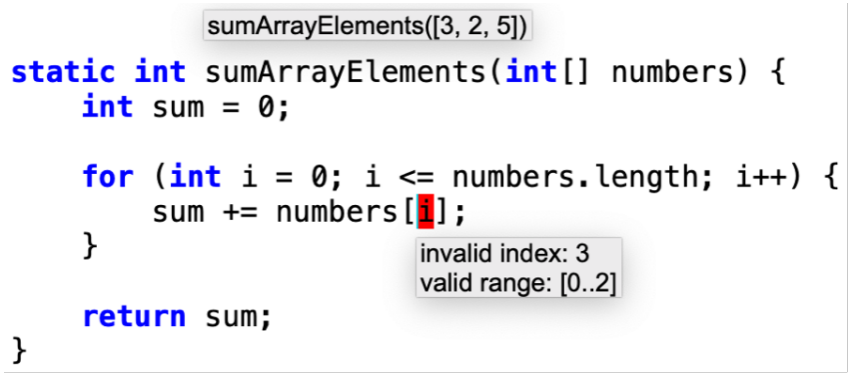


FIGURE 3.18. Error message obtained from an illegal array access.

CHAPTER 4

Implementation

This chapter covers the main implementation aspects of the application, as well as the key technologies used to achieve its features.

4.1. User Interface

The UI of the application was developed using SWT (Standard Widget Toolkit)[1], which is a graphical widget toolkit for Java that provides a set of native tools for creating platform-independent GUIs. Though it was designed for Java, SWT can also be used in a Kotlin application because Kotlin is fully interoperable with Java, enabling seamless use of Java libraries such as this.

One of the priorities regarding the user experience with the application was to display most of the information within the main window in a clear and concise manner. The exception to this principle is the window that prompts users to insert the arguments of a method, which we considered to be most effectively presented as a pop-up window.

4.2. Argument Parsing

In the context of this application, users' input parameters require parsing in two occasions: when prompted to insert arguments in the pop-up window upon calling a method, and when writing a test in the unit test table. Both cases are dealt with in a similar way, and their handling relies on the JavaParser library[2] , which provides an Application Programming Interface (API) for analysing Java source code.

We begin by collecting the list of parameters of the method in question and the list of input arguments. Then, for each parameter, we check for its type using the Strudel API, and, as long as it is valid (i.e., it is one of the types supported by the application), we attempt to parse the corresponding input argument as the specified type using JavaParser. For example, in the case of a method that takes an integer parameter, users' input argument is parsed as an integer literal expression. An analogous process happens for `double`, `char` and `boolean` variable types. In the case of arrays, we use JavaParser to determine the type of the input array and its values. Finally, each value is parsed as the inferred type, failing if any parsing operation is unsuccessful.

As mentioned in the above chapter, our environment requires array inputs to be passed in a format that may seem unusual to a more experienced programmer. To ensure parsing consistency, we perform specific manipulations on the input array arguments to make them compatible with JavaParser. This manipulation occurs as soon as the method is called.

[1] `https://www.eclipse.org/swt/`
[2] `http://javaparser.org`

We use a regular expression to detect array creation patterns and transform them into valid array expressions with the correct prefixes (e.g., the input `[1, 2, 3]` is converted to `new int[] 1, 2, 3`). These adjusted input arguments are then processed according to the described parsing procedure.

4.3. Data Collection

The information displayed in the code tooltips is collected during the execution of each method via a listener provided by the Strudel API. As mentioned in Section 2.2, when users run a method in our tool, its execution is simulated within a Strudel virtual machine. We implemented the listener to monitor specific behaviours relevant to the debugging task. It is activated at key points in an execution, such as when a variable is assigned a value, or a loop performs an iteration. Each of these actions is handled by a distinct method within the listener. Figure 4.1 illustrates how such methods are invoked during the execution process. It is important to note that, for simplicity's sake, the image omits multiple other invocations which would also be triggered during the execution of this sample code.

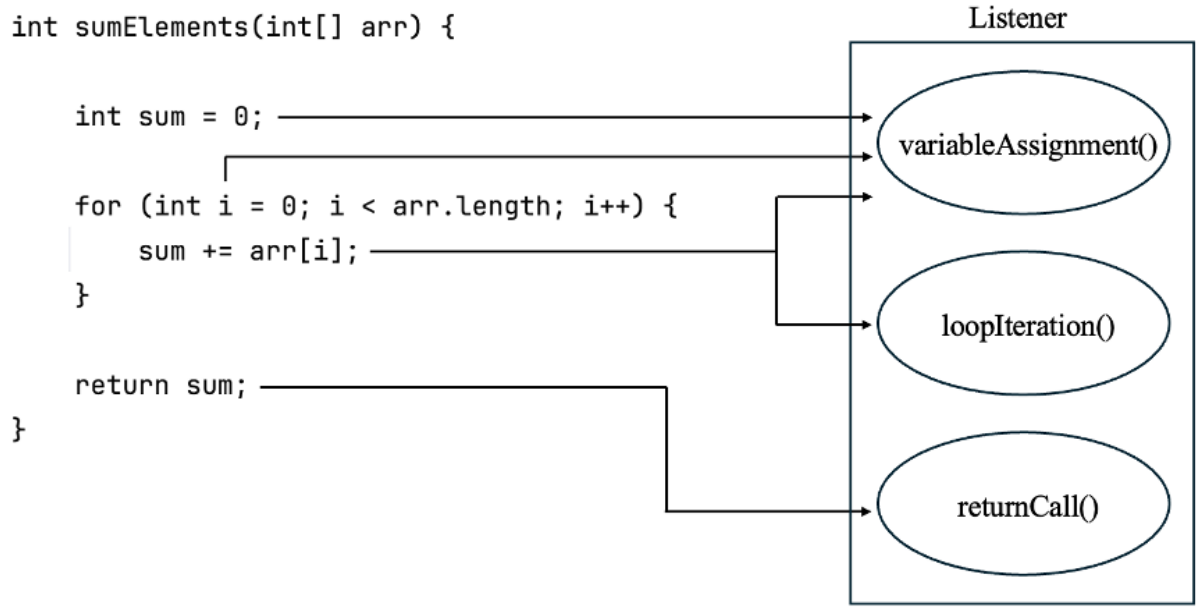


FIGURE 4.1. The listener monitors the execution and invokes methods to handle each action.

Since the execution is simulated, we can access detailed information about any aspect of its process through Strudel objects that represent the characteristics of each program element. For instance, in the `returnCall` method, we can retrieve the structure of the return statement, along with its value, and process this information to present it accordingly in a tooltip.

4.3.1. The Process

The data collection process itself unfolds in three main stages. First, multiple data structures are initialised within the listener to track the program elements of an execution. These

structures are responsible for tasks such as mapping code statements to the corresponding text to be displayed in the tooltips and mapping loop objects to lists representing variable tracings. Then, as the execution proceeds, these structures are incrementally populated as the listener methods are triggered by various events. Finally, after the method's execution ends, the collected data is organised in the form of an object to be accessed within the debugging view. Figure 4.2 is a UML diagram representing the structure of the content stored in this object, referred to as "Execution Data". Certain details that entail more complex implementations have been omitted for simplification. This data type contains the location in the IDE of the corresponding method (`procLocation`), the return value obtained from the execution (`result`), and data structures storing information on variables, loops and expressions that may be inspected (`variableInspections`, `loopTraces` and `expressionTraces`). Also shown in the diagram are two data types which relate to the creation of the `ExecutionData` object. `ExpressionValue` supports the data collection regarding expression tracing, representing an association between an expression and its evaluation at a given moment in the code. Likewise, `VariableTrace` assists on storing information concerning variable tracing, representing a variable's history of values.

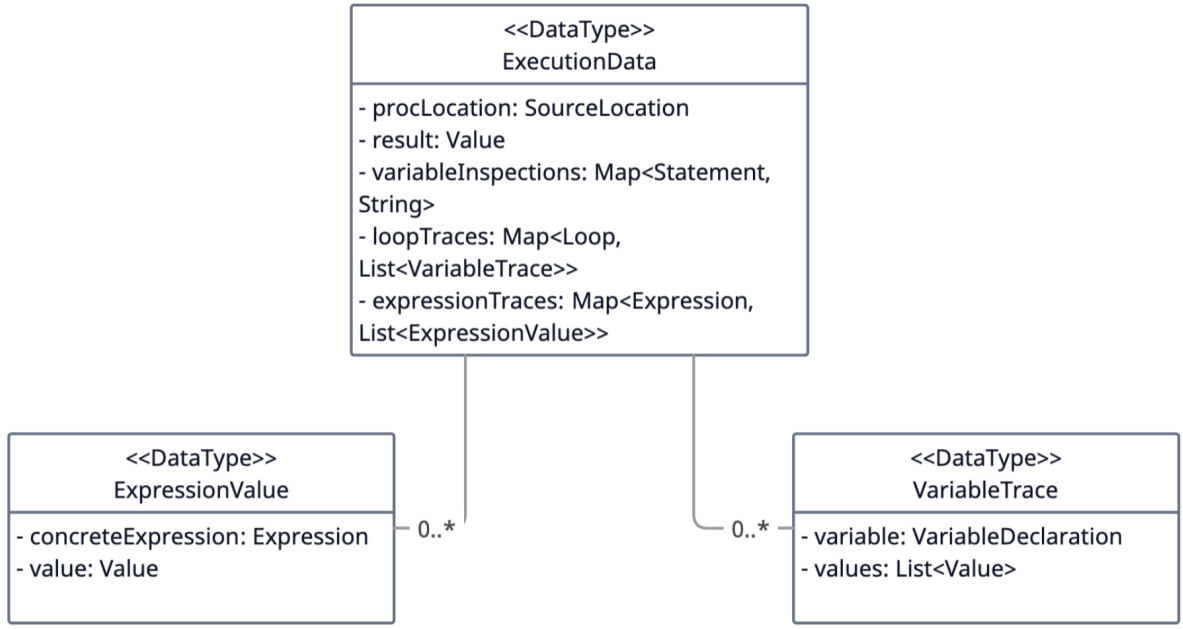


FIGURE 4.2. UML diagrams representing the data types `ExecutionData`, `ExpressionValue` and `VariableTrace`

These data types were built on top of existing structures provided by Strudel. For instance, `Statement`, used in the `variableInspections` map, is an interface encompassing objects that can represent any syntactic unit or instruction that performs a specific action, including assignments, return statements and control flow statements. On the other hand, `Loop`, used in the `loopTraces` field pertains to loop blocks. Strudel's hierarchical approach allows a break down of each element within a method, supplying the means for handling the information needed for our debugging purposes.

The list below comprises every listener method implemented in this project, which populate the components of an `ExecutionData` object, along with a brief description of its behaviour. While Strudel provides the structure for handling multiple other events, we found it unnecessary to implement them within the scope of the tool's debugger.

- `procedureCall` initialises data structures for collecting information in later methods.
- `variableAssignment` traces variable assignment values.
- `arrayElementAssignment` traces array element assignment values.
- `loopIteration` collects information on loop variables before beginning an iteration.
- `loopEnd` collects information on loop variables after the last iteration.
- `expressionEvaluation` traces expression evaluations.
- `returnCall` collects information on the return statement.
- `executionError` collects information on any potential execution error.
- `procedureEnd` processes all information from the method and stores it in an `ExecutionData` object.

The following section covers key concepts that play a major role in the implementation of these listener event methods.

4.3.2. Listener Events

In Section 3.4.1, it is explained how variable roles influence the display style of certain code inspections. This categorisation first occurs in the data collection stage. One listener event method where variable roles are taken in consideration is the "variableAssignment" method, represented in pseudocode in Figure 4.3. The control structure of this method demonstrates the core logic underlying the debugger's operation. It is worth noting that this illustration does not cover every intricacy regarding the method's logic, and this also applies to other events depicted in this section.

In relation to how a variable's behaviour can be represented, we focus on aspects that can influence the way users may interpret the code inspections. For instance, fixed-value variables require minimal processing, as their assigned values can be directly stored as the corresponding tooltip's content. However, a fixed-value variable inside a loop may be different across iterations. We account for those cases with the condition `if (!a.parent.isInLoop)`. This is one of the multiple utility methods provided by the Strudel API. If the condition is true then we can immediately store the tooltip inside the `variableInspections` map. If, however, the variable is indeed inside a loop, we must treat it as a regular variable in a process akin to what is shown in the `else` statement of the `when` structure.

A similar consideration was made for stepper variables, although in this case we considered to be best not to display tooltips when such variables occur inside loops (e.g., an index variable inside a nested loop), as their representation would not be trivial. In any

```
fun variableAssignment(a: VariableAssignment, value: Value) {
    when (a.role) {
        is FixedValue -> {
            if (!a.parent.isInLoop) {
                variableInspections.put(a, value)
            }
            else {
                addVariableTrace(a, value)
                statements.add(a)
            }
        }
        is Stepper -> {
            if (!a.parent.isInLoop) {
                statements.add(a)
            }
        }
        else -> {
            addVariableTrace(a, value)
            statements.add(a)
        }
    }
}
```

FIGURE 4.3. The `variableAssignment` method (pseudocode).

other case, stepper variables are treated in a slightly different manner than the remaining roles. Since only their first and final values are included in the tooltip, considering that intermediate values are self implied, the need for storing every assigned value during the execution, via the method `addVariableTrace`, is eliminated. Therefore, we simply store the statement in which the assignment takes place (`statements.add(a)`) and handle it later in the `procedureEnd` method.

In order to exemplify this procedure, we use the `sumElements` method, depicted in Figure 4.1, as an illustration of how a variable assignment call is handled in a practical scenario, in addition to other event methods which would be triggered during the execution. For this example's sake, we assume that the method was called with the following input argument: $[1, 2, 3]$. In the first line, "`int sum = 0;`", the value 0 is assigned to the integer variable `sum`. This action triggers the `variableAssignment` listener method, which accepts two parameters; the assignment expression, `a` and the value being assigned, `value`. First, we check the variable's role to determine how this instance will be handled. Given that this variable's final value is obtained by an accumulation of previous values, Strudel categorises it as a "gatherer" variable, which directs the method to proceed using the `else` clause. From here, we call the method `addVariableTrace`, passing the parameters needed to create and store an object of `VariableTrace`, representing the content of this variable at this point in the code. From this event, the format of the collected object will resemble the following instance: `VariableTrace(Statement("int sum=0;", 0))`. The statement encapsulating this assignment is also passed to an auxiliary structure `statements` which helps matching each program element to its corresponding tooltip, after which the method

ends. All of this information will later be consulted after the execution ends, to prepare the tooltip that will be displayed in the debugger, in a style that is appropriate for a "gatherer" variable.

As we follow this example's execution, the loop would then iterate over the elements of the input array. In each iteration, several listener methods are triggered as a result of the behaviour exhibited in the code. After declaring and assigning the integer variable `i`, setting off the `variableAssignment` event. A procedure similar to what is described in the above chapter will take place, with the particularity that this is a "stepper" variable, prompting slightly different handling, as previously described.

The condition that follows triggers the `expressionEvaluation` method, illustrated in Figure 4.4. The distinction between the input parameters `e` and `concreteExpression` pertains to the fact that Strudel provides access to both the raw expression as well as the evaluated expression at that moment, which we utilise to better manage the information obtained from this event. Regarding the method's logic, first we determine if the expression is literal, in which case no further processing is required, and, assuming it is not, we collect the respective trace in an `ExpressionValue` object and store it in the `expressionTraces` data structure. In this example, the condition checks if `i`, which has the value 0, is smaller than the length of the array (3). The resulting `ExpressionValue` object would therefore resemble `ExpressionValue("0 < 3", true)`. Along with the evaluated expressions obtained during the next iterations, this information can later be displayed in a tooltip upon inspection, tracing the condition's multiple evaluations throughout the execution.

```
fun expressionEvaluation(
    e: Expression,
    value: IValue,
    concreteExpression: Expression
) {
    if (e is Literal)
        return

    expressionTraces.put(
        e, List<ExpressionValue(concreteExpression, value)>
    )
}
```

FIGURE 4.4. The `expressionEvaluation` method (pseudocode).

Entering the loop iteration itself, the event `loopIteration` will be called, compiling all variable assignments that take place during its course and storing their data in the `loopTraces` structure, which is basis for the creation of the loop inspection table. The above procedures repeat for each iteration, populating the various data structures, until the final iteration, which activates the `loopEnd` event to complete the process.

Upon reaching the return statement, the listener event `returnCall` is triggered, storing information regarding the evaluation of the return expression. In this case, if we sum the input array's elements, we obtain the value 6 $(1 + 2 + 3)$.

Finally, after the execution has terminated, and assuming no runtime errors occurred, the calling of the `procedureEnd` event organises all collected data through an `ExecutionData` object containing the information to be displayed in the tooltips. If users wish to execute multiple other tests of the same method using the unit test table, they are able to inspect each one individually and seamlessly alternate between inspections, due to each being associated to a separate `ExecutionData` instance. The debugging view simply interprets the content of each instance and highlights the code accordingly.

Though not extensively, this section shows the general logic behind the data collection process which supplies the debugging view with tooltips for each program element worthy of being inspected. While other aspects could have been explored, the discussed listener events are the ones we deem best to represent this procedure.

CHAPTER 5

Usability Study with Novice Programmers

To explore **RQ2**, we conducted a preliminary usability study with 6 novice programmers, in order to evaluate the effectiveness of the prototype.

In order to assess the prototype's usefulness, we sought feedback from individuals with matching characteristics to the potential users of the application - crucially, subjects with little programming knowledge or experience. We intended to validate the application's viability and gather constructive criticism regarding its main features, most notably the code inspections.

The following sections describe the testing process, including our method and the obtained results.

5.1. Method

The individuals who volunteered to participate in the study presented mostly similar backgrounds, generally coming from finance and economics related degrees, with the exception of one subject who has a bachelor's degree in data science. Every participant had previously taken at least one programming course using Python, and none had experience in Java. As such, before formally starting a session, we dedicated a few minutes providing an overview of the environment's user interface and addressing any questions about the upcoming procedures. This included short exercises to bring the participants into a programming mindset, such as simple questions on loops and arrays, to review possibly forgotten concepts and illustrating how they materialised in the Java language.

Each session consisted of four tasks of increasing difficulty centred around fixing bugs in four faulty Java methods. Participants were instructed to perform half of the tasks using our application and the other half using the IntelliJ[1] IDE, which is currently one of the most popular coding environments for Java. The four methods involved in the sessions, which can be found in Appendix A, presented the following issues:

(1) The method `factorial` miscalculated the factorial of a number.
(2) The method `isPalindrome` incorrectly checked if a word was a palindrome or not.
(3) The method `sequence` threw an "array index out-of-bounds" exception when attempting to create an array consisting of every integer number between two given parameters.
(4) The method `moveZerosToEnd` threw an "array index out-of-bounds" exception when attempting to move the zeros of an integer array to the end.

[1] `https://www.jetbrains.com/idea/`

The tasks were presented to every participant in the same order. However, half of the group used the prototype to solve tasks 1 and 3, while the other half used it for tasks 2 and 4. The exercises were designed to simulate common mistakes made by beginner programmers, such as off-by-one errors (e.g., intending to iterate over all elements in an array, but missing the last one), incorrect arithmetic logic and array accessing issues. Each method's intended behaviour and implementation was carefully explained to the subjects before moving on to solving the exercises.

We measured the completion time for each task and used this data to calculate the average time spent across both platforms. Using this approach, we could then compare the subjects' average performance on the two environments. A positive outcome would be reflected in shorter average task completion times when using the prototype compared to a conventional IDE.

Additionally, we observed and documented any indications of uncertainty or confusion expressed by participants to evaluate potential usability issues. Although usability may not be numerically measurable, we attempted to gain insights into the subjects' experience with the prototype by noting relevant remarks and feedback during the experiment.

5.2. Results

Table 5.1 presents every subject's completion times for all tasks. Values in bold correspond to instances where the prototype was used, while values in plain text represent tasks completed with the conventional tool. Below the individual task completion times are the average times and standard deviation in each environment, followed by the time difference percentage. A negative percentage indicates that the subjects completed the tasks faster, on average, using the prototype. Positive results reflect the opposite scenario.

TABLE 5.1. Test results.

Subject/Task	`factorial`	`isPalindrome`	`sequence`	`moveZerosToEnd`
Subject 1	0:20 min	**0:35 min**	0:58 min	**1:29 min**
Subject 2	**1:50 min**	1:13 min	**5:35 min**	4:21 min
Subject 3	2:57 min	**2:09 min**	5:05 min	**3:34 min**
Subject 4	**1:27 min**	1:28 min	**3:04 min**	4:36 min
Subject 5	1:37 min	**2:25 min**	6:33 min	**5:12 min**
Subject 6	**2:46 min**	1:46 min	**4:50 min**	3:09 min
Average time (IntelliJ)	1:38 min	1:29 min	4:12 min	4:02 min
Std Deviation (IntelliJ)	1:18 min	0:16 min	2:54 min	0:47 min
Average time (prototype)	**2:01 min**	**1:43 min**	**3:03 min**	**3:25 min**
Std Deviation (prototype)	**0:40 min**	**0:59 min**	**1:18 min**	**1:52 min**
Time difference (%)	+23.5%	+15.7%	-27.4%	-15.3%

Although the number of participants is relatively small, this study showed interesting trends that gave us valuable insight into potential use cases for the prototype.

It is worth noting the time discrepancy between some of the subjects can be attributed to the varying degrees of comfort the individuals had with programming. Although their

academic experience was largely similar, some subjects generally enjoyed the programming courses they took during their degrees more than others. These individuals were generally more comfortable with basic concepts even if years had passed since they took the courses, leading to overall faster times.

Regarding the time difference between environments, it is clear that the prototype did not entail better results in tasks 1 and 2. On the other hand, in tasks 3 and 4 we find the exact opposite. Looking into the reasons behind these observations, one could note that the first two tasks revolved around simple arithmetic problems and basic programming structures. In reference to the "factorial" task, one participant commented after the session that as soon as he ran the method with some input N and saw that it had computed the result for an input $N - 1$, it was very clear to them what the problem was. They immediately realised that the loop had performed one iteration less than what it should have. Perhaps the simplicity of these problems may not justify resorting to a debugging environment. The same cannot be said for the last two tasks, as they concerned more complex structures and errors. A method that throws an exception typically intrigues beginner more than an incorrect result. In tasks 3 and 4, some participants spent much of their time first trying to understand what exactly the issue was, and only then attempting to fix it. The code inspections and error messages shown in the prototype seem to have helped some of the subjects, however, this claim would require a larger and therefore more robust study to be accurately verified.

Overall, the study presented in this chapter places the focus of future tests in assessing student's performances in situations where the behaviour of a program may not be trivial, such as more complex algorithms, akin to those typically taught in a course on programming algorithms and data structures.

CHAPTER 6

Conclusions

The work presented in this dissertation stemmed from the observation of existing pedagogical programming visualisation tools and their shortcomings as beginner friendly tools that assist students' development of coding skills. We proposed a prototype which aims to tackle common issues manifested by students in the context of introductory programming courses, most notably, tracing the history of variables' values throughout the execution of a program. The proposed Research Questions **RQ1** and **RQ2** guided the development of the prototype in its entirety, serving as validation mechanisms for the design choices of the application.

Regarding Research Question **RQ1**, we consider the prototype to be successful in presenting variable and expression tracing features. Our proposed environment demonstrates a valid approach for illustrating such concepts, emphasising those that are tipically more problematic to beginner students. For instance, features such as loop inspections complement the tool in providing the means for a thorough analysis of a program execution, particularly in scenarios that rely on fundamental data and control structures, commonly found in the curriculum of introductory courses. Additionally, role specific inspections suggest the behaviour of certain variables whose tasks in the context of a program can often be reproduced in different exercises, without explicitly referring to such variables by a role name. The expression tracing features can further assist users in understanding a program's information flow, through inspections that reveal the evaluation of conditions at different points in the code, or the parts that compose complex variable assignment expressions.

When it comes to Research Question **RQ2**, the conducted study on subjects that are inexperienced in programming offers valuable insights on the potential experience encountered in a real world scenario. Our findings reveal that the tracing features had a positive impact in the subjects' performance in the two most complex tasks, encouraging us to believe that the proposed debugger could benefit students in an introductory programming class context. Given the participants' limited coding experience, the debugger was helpful in pointing to the direction of the solution.

6.1. Limitations and Future Work

The limitations of this dissertation can be categorised into two main topics:

- Shortcomings of the prototype;
- Scale and limitations of the usability study.

Regarding the first category, the prototype encompasses many of the core concepts faced by students in introductory programming courses. However, it lacks coverage of some aspects which are commonly found in the curriculum of courses of this nature. For instance, we were not able to implement support for multi-dimensional arrays, a feature which typically provides challenging problems to inexperienced students. Additionally, Strudel's limited support for Java types restricts users from experimenting with collections. Also, the application allows for a single class to be explored at a time, due to the application's single file setup. Although the prototype is not tailored to object-oriented programming, we hope to expand its features in future iterations to better accommodate these concepts, providing a more comprehensive environment for students.

In relation to the usability study, we believe that additional research is recommended to solidify the evidence we gathered from our experiment. While the results are promising and suggest that the prototype can be useful in diverse situations, further studies with a larger participant pool and a broader range of tasks are needed to establish the efficacy of such an environment in real world scenarios.

In conclusion, while this dissertation leaves room for further exploration, we consider the presented product to represent a meaningful contribution towards pedagogical debugging research. It serves both as a proof of concept for the Strudel library's capabilities of facilitating the creation of behavioural inspection based debugging environments, and as a potentially valuable tool for novice students, aiding in their programming learning path.

References

[1] J Buchan, L Li, and S. G. Macdonell. Causal factors, benefits and challenges of test-driven development: Practitioner perceptions. *18th Asia-Pacific Software Engineering Conference*, 2011.

[2] P Byckling and J Sajaniemi. Roles of variables and programming skills improvement. *ACM SIGCSE Bulletin*, 38:413–417, 2006.

[3] C Desai, D Janzen, and K Savage. A survey of evidence for test-driven development in academia. *ACM SIGCSE Bulletin*, 40:97–101, 2008.

[4] J Edwards. Example centric programming. *ACM SIGPLAN Notices*, 39:84–91, 2004.

[5] P Gestwicki and B Jayaraman. Methodology and architecture of jive. *SoftVis '05: Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 95–104, 2005.

[6] P Gries, V Mnih, J Taylor, G Wilson, and L Zamparo. Memview: A pedagogically-motivated visual debugger. *Proceedings Frontiers in Education 35th Annual Conference*, 2006.

[7] B Hartmann, D MacDougall, J Brandt, and S. R. Klemmer. What would other programmers do? suggesting solutions to error messages. *CHI '10: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2:1019–1028, 2010.

[8] T Jenkins. On the difficulty of learning to program. *3rd Annual LTSN-ICS Conference, Loughborough University*, pages 53–58, 2002.

[9] M Kuittinen and J Sajaniemi. Teaching roles of variables in elementary programming courses. *ITiCSE '04: Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 57–61, 2004.

[10] M Kölling, B Quig, A Patterson, and J Rosenberg. The bluej system and its pedagogy. *International Journal of Phytoremediation*, 13:249–268, 2003.

[11] E Lahtinen, K Ala-Mutka, and H.-M. Järvinen. A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin*, 37:14–18, 06 2005.

[12] A Moreno, N Myller, E Sutinen, and M Ben-Ari. Visualizing programs with jeliot 3. *AVI '04: Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 373–376, 2004.

[13] G Pothier and É Tanter. Back to the future: Omniscient debugging. *IEEE Software*, 26:78–85, 2009.

[14] Eric Allen Robert, Robert Cartwright, and Brian Stoler. DrJava: A lightweight pedagogic environment for Java. In *SIGCSE Bulletin and Proceedings*, pages 137–141. ACM Press, 2002.

[15] J Sajaniemi and M Kuittinen. Program animation based on the roles of variables. *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 7–16, 2003.

[16] A. L. Santos. Aguia/j: a tool for interactive experimentation of objects. *ITiCSE '11: Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, pages 43–47, 2011.

[17] A. L. Santos and H. S. Sousa. Pandionj: A pedagogical debugger featuring illustrations of variable tracing and look-ahead. *ACM International Conference Proceeding Series*, pages 195–196, 2017.

[18] V Vainio and J Sajaniemi. Factors in novice programmers' poor tracing skills. *ACM SIGCSE Bulletin*, 39:236–240, 2007.

[19] A Yadin. Reducing the dropout rate in an introductory programming course. *ACM Inroads*, 2:71–76, 2011.

APPENDIX A

Usability Testing Task Code

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    int result = 1;

//  correct:
//  (int i = 2; i <= n; i = i + 1)

    for (int i = 2; i < n; i = i + 1) {
        result = result * i;
    }
    return result;
}
```

FIGURE A.1. Factorial method implemented in Java.

```
boolean isPalindrome(int number) {
    int original = number;
    int reversed = 0;
    while (number != 0) {
        int digit = number % 10;

//      correct:
//      reversed = reversed * 10 + digit;

        reversed = reversed + digit * 10;
        number = number / 10;
    }
    return original == reversed;
}
```

FIGURE A.2. Method in Java to check whether a given integer is palindromic.

```
int[] moveZerosToEnd(int[] nums) {
    int insertPos = 0;
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] != 0) {
            nums[insertPos] = nums[i];
            insertPos++;
        }
    }
    while (insertPos < nums.length) {

//      correct:
//      insertPos++;
//      nums[insertPos] = 0;

        nums[insertPos] = 0;
        insertPos++;
    }
    return nums;
}
```

FIGURE A.3. Method in Java to move all integers equal to 0 in an array to the end.

```
int[] sequence(int from, int to) {
    int size = to - from + 1;
    int[] seq = new int[size];
    int i = 0;
    while (i < size) {

//      correct:
//      seq[i] = from + i + 1;

        seq[i + 1] = from + i + 1;
        i++;
    }
    return seq;
}
```

FIGURE A.4. Method in Java to fill an array with an integer sequence.