INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Illustration of Java execution errors for beginner programmers

Diogo Alexandre Rodrigues de Sousa

Master in Telecommunications and Computer Engineering

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

Co-Supervisor:
Doctor Maria Cabral Diogo Pinto Albuquerque,
Assistant Professor,
Iscte - Instituto Universitário de Lisboa

October, 2020

Acknowledgments

This thesis represents one of the greatest achievements in my academic life, which could not be accomplished alone. Through the writing of this dissertation, I received a lot of assistance and support.

I would like to start by thanking my supervisors, André L. Santos and Maria Pinto Albuquerque. Their constant guidance and willingness to help whenever necessary has helped me overcome many difficulties. Without them, the thesis in its present form would not be possible.

Next, I would like to thank my colleague Ricardo Castro for accompanying me during the course of this work.

Last but not least, I would like to thank my family and my friends for the motivation they provided and encouraging me to never give up pursuing my goal of completing this work.

Resumo

A programação é um assunto aprendido por estudantes de todo o mundo. Muito estudantes encontram programação pela primeira vez em aulas introdutórias e enfrentam muitos conceitos novos que nunca tinham visto antes. Uma das principais dificuldades que encontram são os erros de execução. As aulas introdutórias de programação com a linguagem Java não abordam os erros de execução de uma forma clara e fácil para os estudantes compreenderem o que são e como os evitar. A stack trace do Java não é fácil de decifrar sem conhecimento e experiência prévios sobre os vários tipos de exceções e não é fácil de descobrir onde os erros se localizam. Existem diversas ferramentas de programação pedagógicas que fornecem melhores explicações que a stack trace do Java, aproveitando os papéis das variáveis e utilizando ilustrações, mas ignorando exceções no processo.

A abordagem para esta dissertação foi desenvolver uma ferramenta de programação pedagógica que se focasse em erros de execução. Utilizando texto e ilustrações, a ferramenta ajuda os estudantes dando-lhes melhores explicações sobre as exceções e facilitando o processo de descobrir que parte do código desencadeou o erro de execução. Os papéis das variáveis foram também implementados na ferramenta, para ajudar os estudantes ao dividir variáveis em diferentes casos de utilização de forma a explicar melhor como encaixam no código. Para determinar se a ferramenta é eficaz foi realizado um estudo que envolveu vários estudantes, no qual estes realizaram alguns exercícios para determinar se a ferramenta é eficaz na assistência a erros de execução.

Palavras-chave: papéis das variáveis, erros de execução, ferramentas de programação pedagógicas, visualização de programas

Abstract

Programming is a subject that is learned by students all around the world. Many students encounter programming for the first time in introductory classes and face many new concepts they have never seen before. One of the main difficulties they encounter concerns understanding execution errors. Introductory classes do not address execution errors in a clear way that makes it easier for students to comprehend what they mean and how to avoid them. The Java stack trace is not easy to decipher without previous knowledge and experience on what each type of exception means and not specific enough for a beginner to pinpoint where the problem occurs. Several pedagogical programming tools exist that provide better explanations than the stack trace alone, taking leverage of variable roles and illustrations but neglecting exceptions in the process.

The approach of this thesis was to develop a pedagogical tool that focused in providing explanations of execution errors. Using text and illustrations, the tool helps students by giving them better explanations of exceptions and facilitating the process of discovering what part of the code triggered the execution error. Roles of variables are also implemented, helping students by dividing variables into different use cases to better explain how they fit into the code. To determine whether the tool is effective, a study was carried out involving students in which they carried out some exercises to determine whether the tool is effective in assisting with execution errors.

Keywords: variable roles, execution errors, pedagogical programming tools, program visualization

Contents

Acknowledgments iii

Resumo v

Abstract vii

List of Figures xi

List of Tables xiii

1. Introduction 1

1.1. Motivation 1

1.2. Research questions 1

1.3. Objectives 2

1.4. Research Method 2

2. Related Work 5

2.1. Minimal guidance versus guided instruction in learning 5

2.2. Roles of Variables 6

2.3. Pedagogical programming tools 7

2.3.1. PlanAni 8

2.3.2. BlueJ 9

2.3.3. PandionJ 11

3. Enhancing explanation of Java execution errors 15

3.1. Illustrating Java Execution Errors 15

3.2. Roles of Variables 16

3.2.1. Fixed value 16

3.2.2. Stepper 17

3.2.3. Array Index Iterator 19

3.3. Graphical Interface 20

3.4. Architecture 26

3.4.1. Static Analysis 27

3.4.2. Dynamic Analysis 28

3.4.3. Explanation and Illustration 29
4. User Study 33
4.1. Method 33
4.2. Results and analysis 35
4.3. Threats to validity 40
5. Conclusions and Future Work 41
References 43
Appendix A. Methods created for the experiment 45
Appendix B. Experiment questions 49

List of Figures

Figure 1.1 - DSRM Process Model (Figure 1 from [2]) 2
Figure 2.1 - Image of the Eclipse Debugger when an ArrayIndexOutOfBoundsException occurs 8
Figure 2.2 - Exception at the end of Figure 2.1's example 8
Figure 2.3 - PlanAni animating the first four numbers of Fibonacci's Sequence with Java 9
Figure 2.4 - BlueJ's debugger displaying the same example used in Figure 2.1 11
Figure 2.5 - BlueJ displaying the same Exception as shown in Figure 2.2 11
Figure 2.6 - PandionJ displaying the same example used in Figure 2.1 12
Figure 2.7 - PandionJ when the exception displayed in Figure 2.2 happens 13
Figure 3.1 - Early Draft of an array illustration 16
Figure 3.2 - Fixed Value Verification Diagram 17
Figure 3.3 - Stepper Verification Diagram 19
Figure 3.4 - Array Index Iterator Verification Diagram 20
Figure 3.5 - Prototype's graphical interface 21
Figure 3.6 - Prototype's graphical interface after executing code 22
Figure 3.7 - Explanation and Illustration Area from Figure 3.6 23
Figure 3.8 - Array Illustration Example from Figure 3.7 24
Figure 3.9 - Shortened Array Example 25
Figure 3.10 - Prototype's interface with a method involving matrices 25
Figure 3.11 - Matrix illustration example with different row sizes and horizontal array error 26
Figure 3.12 - Code analysis process diagram 27
Figure 4.1 - Interface when showing regular Java stack trace 35
Figure A.1 - First Method - Return sum of all numbers in array 45
Figure A.2 - Second method - Return an array with all the natural numbers up to n 45
Figure A.3 - Third method - Return index of the last occurrence of an integer in an array 46
Figure A.4 - Fourth Method - Invert array 46
Figure A.5 - Fifth Method - Scale a matrix by multiplying each number by n 47
Figure A.6 - Sixth Method - Return a transposed matrix by swapping lines with columns and vice versa 47
Figure B.1 - First two questions of method one 49
Figure B.2 - Multiple-choice question for the first method 49
Figure B.3 - Options for the multiple-choice question of the first method (Answers: 1, 3) 50
Figure B.4 – Last question of the first method (Answers: 1) 50

Figure B.5 - Multiple-choice question for the second method 51
Figure B.6 - Options for the multiple-choice question of the second method (Answers: 2) 52
Figure B.7 - Last question of the second method (Answers: 3) 52
Figure B.8 - Multiple-choice question for the third method 53
Figure B.9 - Options for the multiple-choice question of the third method (Answers: 1, 4) 54
Figure B.10 - Last question of the third method (Answers: 2, 3) 54
Figure B.11 - Multiple-choice question for the fourth method 55
Figure B.12 - Options for the multiple-choice question of the fourth method (Answers: 2) 56
Figure B.13 - Last question of the fourth method (Answers: 1, 3, 4) 56
Figure B.14 - Multiple-choice question for the fifth method 57
Figure B.15 - Options for the multiple-choice question of the fifth method (Answers: 3) 58
Figure B.16 - Last question of the fifth method (Answers: 2) 58
Figure B.17 - Multiple-choice question for the sixth method 59
Figure B.18 - Options for the multiple-choice question of the sixth method (Answers: None) 60
Figure B.19 - Last question of the sixth method (Answers: 1, 3, 4) 60

List of Tables

Table 2.1 - Roles of variables (Table 1 from [1]) 6
Table 4.1 - Results for the open questions 36
Table 4.2 - Results for code multiple-choice question 37
Table 4.3 - Results for the general content multiple-choice question 38
Table 4.4 - Average time necessary to answer questions (minutes:seconds) 39

1. Introduction

1.1. Motivation

Programming is a difficult task for many students [1]. Learning how to program for the first time can be very hard and frustrating if the proper follow up is not provided. Many students, especially beginners, face problems in their learning process that prevent them from advancing into more complicated tasks.

Execution errors are a case where students can find it difficult to isolate and, more importantly, understand what the problem is, how to fix and avoid it. Java's call stack trace provides basic information of where the problem happened but does so in a convoluted way that is not beginner-friendly and can lead to frustration. A lot of times, the student requires help from a third party, like a teacher or one of his colleagues, in order to overcome such errors and understand why they happen. One problem of requiring help is if the help the student gets does not explain him in a good way what the problem is, it may leave him without full knowledge of the situation and fail to fix it in the future.

Regular IDEs[1] are tailored for professionals, providing maximum efficiency and speed and making a trade off with ease of use. They only provide the default exception message when an exception occurs, and when more advanced information is required a debugger is used, which again is not easy to use and for a beginner. Pedagogical tools exist to aid students at the start of their journey and help tackle different aspects of learning. The problem with these tools is that they do not focus on execution errors, offering only the normal default exception message that may be insufficient for a beginner to decipher. Developing a tool that can fit the role of helping novices learn about execution errors could bring good benefits to the table, by increasing their knowledge of the language and develop their execution error solving skills.

1.2. Research questions

Execution errors will be the focus of this thesis. As such, the research questions are targeted at the explanation of Java execution errors and the interaction between students and the pedagogical environment itself. Taking this into account, the research questions are as follows:

[1] IDE - Integrated Development Environment

- R1: How to improve the explanation of Java execution errors in pedagogical environments?

- R2: Can student comprehension of Java execution errors improve when using an enhanced pedagogical environment?

1.3. Objectives

The aim of this work is to develop a representation and explanations that will help students understand execution errors and improve their knowledge in the Java language. A tool was developed with the focus of achieving this goal. The main objectives of this work are as follows:

- Develop a pedagogical tool that provides more elaborate execution error explanation, when compared to the conventional Java stack trace.

- Investigate how the provided error explanations help students during their learning process.

1.4. Research Method

The research method of this work is design science research [2]. The use of this method will provide a systematic way of producing a working system, using the steps shown in Figure 1.1.

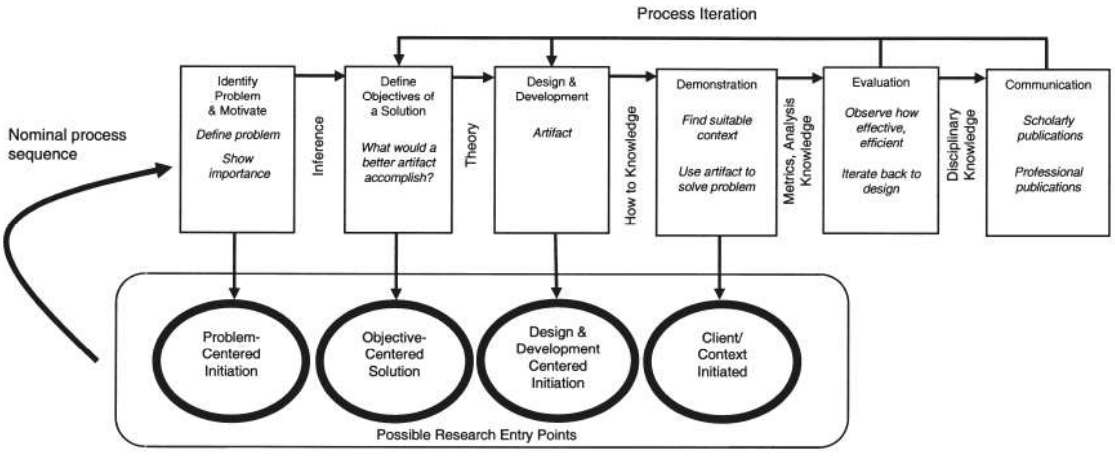

Figure 1.1 - DSRM Process Model (Figure 1 from [2])

The first phase of this research method is the identification of the problem and definition of objectives. In this phase, aspects like previous work done for the studied subject and possible

implementations as well as technologies and methods that may be required to implement and achieve this work's goal are defined.

After the definition of objectives, the second phase involves designing and developing a prototype. This includes tasks like choosing algorithms to use, designing the core system architecture and laying out a graphical interface.

Afterwards, the third phase of the research method involves the implementation of the prototype. In this phase the previously designed components are merged and implemented in order to make a functioning prototype of the system.

The fourth phase is to test the operation of the previously implemented design. This phase involves comparing the current state of the work and observe if it solves the problems of the first phase. If any problem is found, then the necessary modifications are done in order to obtain a finished prototype.

The fifth and final phase is the evaluation phase, in which the final prototype is tested and evaluated for any possible problems that it may have.

Finally, the sixth phase, known as the communication phase, in which the work is published in scholarly publications or similar.

The provided method gives a systematic way of producing a working prototype for a system. All the objectives should be achieved in order to produce a valid prototype. If after the Evaluation a problem is identified, the second, third and fourth phases are repeated until no problem can be found and all objectives are fulfilled.

2. Related Work

2.1. Minimal guidance versus guided instruction in learning

The learning process of humans is very memory dependent. Our long-term memory is what enables us to be skilled in a certain area and be able to do it often unconsciously [3]. The more information is stored in the long-term memory related to a certain subject the more competent a person is in that subject, which means altering long-term memory is the objective of all learning processes.

Before being able to store information in the long-term memory, all information must go through the short-term memory, also known as working memory [3]. All processing is done in this location, which means even knowledge stored in long-term memory must go to working memory in order to be used. Short-term memory is rather limited when dealing with new information, but these limitations do not apply to information acquired previously and stored in long-term memory [3]. According to Miller [4], working memory is limited to a very small number of elements, seven plus or minus two. Peterson and Peterson [5] wrote that knowledge stored in working memory and not trained can be forgotten in just 30 seconds. Moreover, if instead of only storing information, as assumed in the previous works, the information is being processed, it is possible to assume that the number of items may be much less, two or three [3].

Traditional methods of learning often use minimal guidance as a basis for student development, attempting to stimulate development by telling the student to research and complete the task alone or at least with minimal help. In contrast, guided instruction provides the student with examples on how to complete the task at hand and providing help if needed for the student to improve its knowledge [3]. Minimal guidance tries to develop the student ignoring that working memory is limited when dealing with new information. The processing required when trying to learn new knowledge mitigates the possible success of minimal guidance because the student's working memory cannot cope with all the information [6].

Developing a tool with the guided instruction explained above could help students identify problems with the code they are trying to build in a more successful way. In the context of this work, the focus would be in execution errors and helping demonstrate when they happen and how to fix them. Novices could take leverage of this situation by receiving a visual aid when trying to debug problems in order to better stimulate their working memory and provide a solid base for long-term memory to better assimilate the information.

2.2. Roles of Variables

Variables are one of the most fundamental concepts in programming. They are used to store data values. Students struggle with understanding how the various entities that form a program can be used and connected together to form useful and working code [1]. Replacing traditional learning techniques with variable roles can improve the student's knowledge and understanding of variables and how they work in certain contexts. Only ten roles are required to cover 99% of variables used in beginner level programs, displayed in Table 2.1 [7]. The introduction of roles in learning courses provides students with a deeper understanding of the data flow and function of a program, enabling them to process information similarly to good programmers [1]. As such, the use of roles shows positives outcomes when introduced in beginner courses [1].

Table 2.1 - Roles of variables (Table 1 from [1])

Role	Informal description
Fixed Value	A variable initialized without any calculation and not changed thereafter.
Stepper	A variable stepping through a systematic, predictable succession of values.
Follower	A variable that gets its new value always from the old value of some other variable.
Most-recent holder	A variable holding the latest value encountered in going through a succession of values, or simply the latest value obtained as input.
Most-wanted holder	A variable holding the best or otherwise most appropriate value encountered so far.
Gatherer	A variable accumulating the effect of individual values.
Transformation	A variable that always gets its new value with the same calculation from values of other variables.
One-way flag	A two-valued variable that cannot get its initial value once its value has been changed.
Temporary	A variable holding some value for a very short time only
Organizer	An array used for rearranging its elements.

M. Kuittinen and J. Sajaniemi conducted an experiment to test the effects of the introduction of roles in introductory programming course [8]. In order to do a proper evaluation, students were

divided into three different groups: the first one was instructed in a traditional way which did not involve any kind of role learning; the second group utilized roles in the learning process; the last group used roles together with an animator based on them when working in exercises. The animator used was PlanAni, which is explained further in Section 2.3.1. According to Pennington [9], two types of knowledge exist, surface knowledge and deep knowledge. Surface knowledge represents knowledge that is available by simply reading the program's code. Deep knowledge represents knowledge that cannot be read, but instead must come from the correct interpretation of the program's function and how data is handled. After completion of the experiment, the results observed showed that while the animation group had more trouble dealing with surface knowledge, the opposite happened when dealing with deep knowledge. Moreover, the animation group showed better comprehension of deep program structures compared to the roles only group. This phenomenon occurs because roles provide a meaning to each variable of a program, thus helping students understanding how the code works in a deeper level and increasing the accessibility of deep program knowledge. Additionally, the animation group received further benefit from the use of a program animator, which provided a visual aid in understanding how roles behave when using them in exercises.

Introducing variable roles as been proven successful, as explained above, in helping students build better knowledge and comprehension of coding and its internal structure. In order to maximize effectiveness of the learning process, the tool that is going to be developed for the purpose of this work can be further improved using this information. Combining what was wrote in the previous point with the data explained here, it is possible to enhance the tool in such a way that merging these two points can benefit the effectiveness and usefulness of such tool and help students maximize their learning potential. In addition to what a visual aid can offer, variable roles can be introduced in such a way that trying to understand what is being explained by the visual aid is easier and much more intuitive.

2.3. Pedagogical programming tools

When starting to learn how to program, novices have a hard time trying not only to learn the language, but also to acquire a good workflow with the Integrated Development Environment (IDE) they are going to use. The most common IDEs for Java are Eclipse, NetBeans and IntelliJ. Although these tools offer a very good value for an experienced Java developer, they can be overwhelming for a newcomer to learn how to use. When Exceptions start occurring, a developer receives feedback of what is happening from Java's Stack Trace, which contains the location of where the problem occurred and what Exception has been thrown. For a beginner, the information that is given by the Java stack

trace is convoluted and does not offer short and precise information about what happened and how or even what to fix. Another way of discovering problems with code is through a debugger. Debuggers are essential tools for understanding and fixing problems that exist in the code a developer has written. Each of the IDEs specified offer a debugger that is hard for a novice to use when trying to fix problems with the code he wrote.

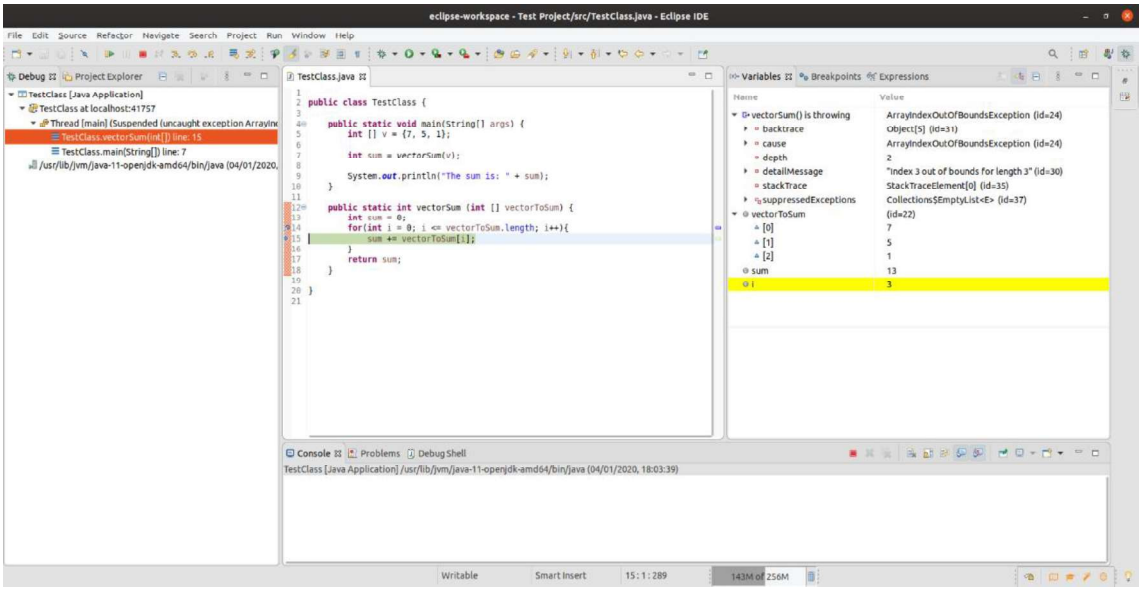

Figure 2.1 - Image of the Eclipse Debugger when an ArrayIndexOutOfBoundsException occurs

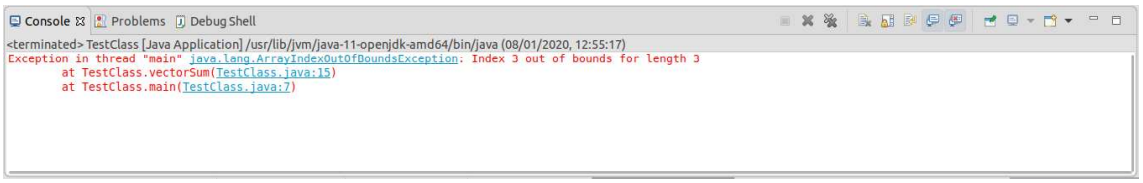

Figure 2.2 - Exception at the end of Figure 2.1's example

2.3.1. PlanAni

PlanAni is a software designed to aid novices in learning programming by using animations and with a heavy focus in roles of variables [10]. It supports several languages, including Java. PlanAni is not capable of animating programs made by users. Instead, all examples must be constructed manually for each program, which involves making the animation commands individually. For each line in the animated program, approximately five lines of animation are required [10].

Roles are represented with images in the animations, for example, a stepper is represented by footprints with the current value in the centre, previous values in one side, next values on the other side and an arrow representing the direction in which the values are going. Arrays are represented

with an image of the role for each element of the array. The images that represent each of the roles were designed taking into consideration the properties of each individual role in which the informal descriptions are described in Table 2.1. In order to test how effective PlanAni is in helping students learn how to construct proper programs, an experiment was made which was described in Section 2.2.

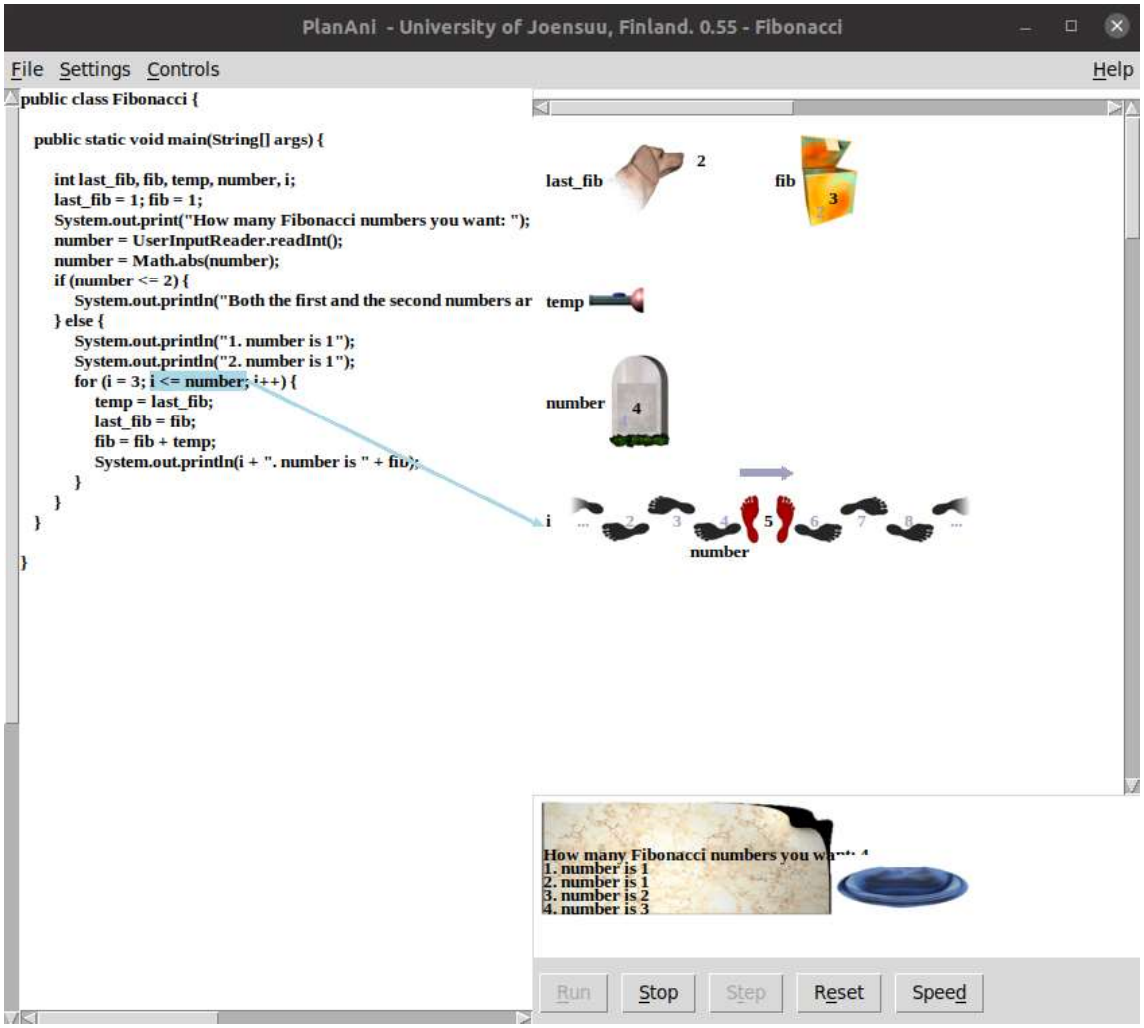

Figure 2.3 - PlanAni animating the first four numbers of Fibonacci's Sequence with Java

Roles of variables can be useful when trying to help debug execution errors. Showing the user which role each variable is can aid in trying to demonstrate how to evaluate the state the program is when the error occurs and help in discovering the culprit and fixing the problem. Using variables roles is easier to explain and demonstrate then using a more traditional method if roles can be discovered and displayed automatically by the tool.

2.3.2. BlueJ

BlueJ is an IDE designed to introduce beginners to the object-oriented nature of Java. It was created in a time where it was believed that object-oriented programming was harder than procedural

programming. The creators of BlueJ had the hypothesis that object-oriented programming was not harder, but the fact that no tools were fit for the job of properly teaching programmers how to use objects properly [11] [12]. The IDE has three major key points in its design: interaction, visualisation and simplicity.

The way BlueJ's graphical interface is designed allows for a much easier comprehension of how objects connect. When a novice sees the class structure displayed he can easily understand that a class is not just some functions that do something, but objects connected together and cooperating with each other [11]. In order to achieve this kind of visualisation, BlueJ adopted an UML (Unified Modelling Language) class diagram that shows the hierarchy of objects in a clear way suited for beginners [12].

BlueJ's interaction allows a student to explore every detail of code in an independent and instinctive way. Simply right clicking in a class allows the user to create an object via one of the constructors or executing a method by itself. The result of said execution is sent to a result dialogue and any resulting new objects can be stored in the object workbench. The workbench is used to store objects that can later be used in methods [11]. This kind of interaction allowed for students to experience with small-scale interactions and build a better comprehension of how objects work on the inside [12].

Finally, BlueJ offers a simple experience for students by removing unneeded features present in more advanced IDEs and focusing more in one specific task, helping novices learn object-oriented programming [11]. This helped students by reducing the burden of their initial interaction with a development environment and helping them focus in building better programming knowledge [12]. Despite the effort of BlueJ's developers, teachers did not use the IDE in the way they wanted. Teachers took advantage of the simplicity that it offered but did not alter their methods in order to explore the potential BlueJ offered for the student [12].

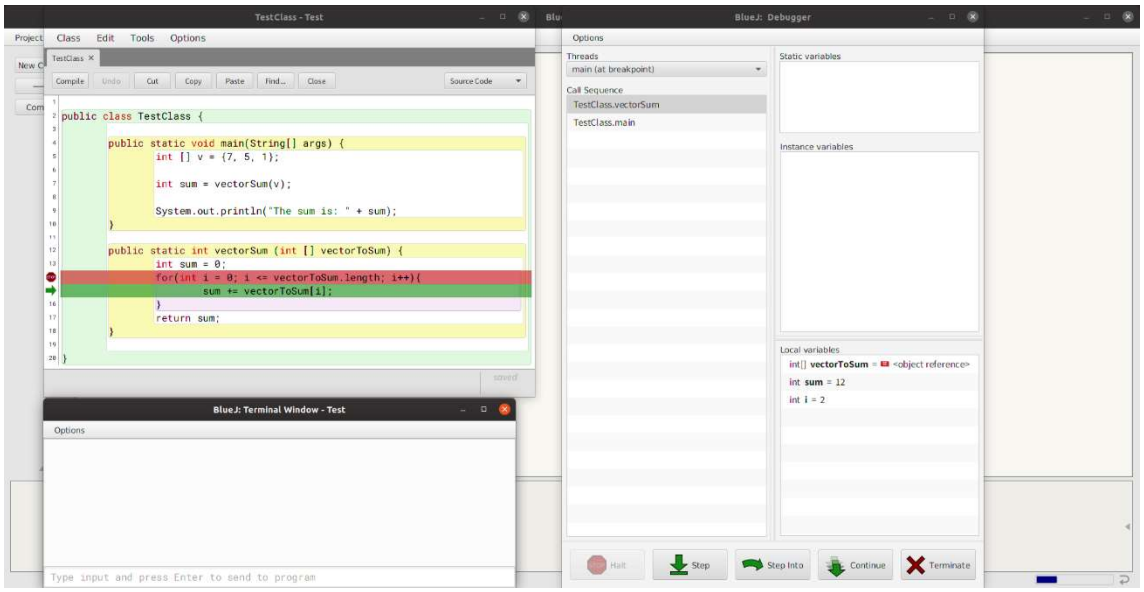

Figure 2.4 - BlueJ's debugger displaying the same example used in Figure 2.1

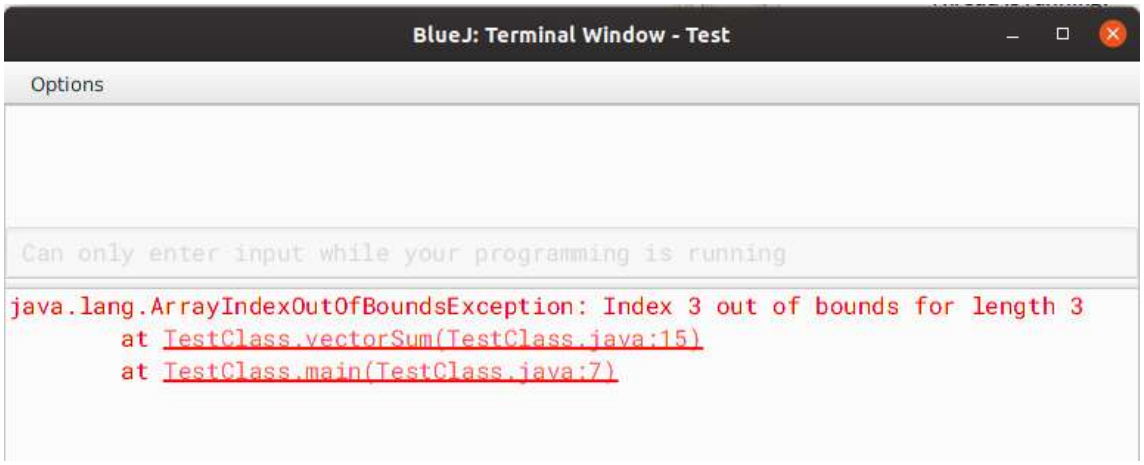

Figure 2.5 - BlueJ displaying the same Exception as shown in Figure 2.2

BlueJ offers a much more intuitive experience for beginners to start programming. It allows for experimentation and easier interaction with objects that already exist or are created by the student itself. Even with these aids, BlueJ does not help the student dealing with execution errors. As shown in Figure 2.4 and Figure 2.5, the debugger by itself may offer more information than for example the standard eclipse debugger but does not offer an explanation to the student that allows him to better understand what is happening and how can the error be avoided, which in turn requires the student to seek outside help.

2.3.3. PandionJ

PandionJ is a pedagogical debugger designed for novices. It contains features present in normal debuggers but provides further enhancements such as the inclusion of variable roles and displaying

relationships between the variables [13]. It was created as an extension of Eclipse and takes advantage from the prebuilt debugger engine Eclipse provides, while also using static analysis of code in order to obtain information of variable roles and displaying a beginner friendly user interface.

The creation of PandionJ involved a previous study about how programming teachers illustrated variables and control flow [14]. Several teachers were tasked with exercises, in which they had to explain the execution of a certain method. They were free to draw them. After every teacher completed the experiment, their explanations and drawings were examined and reviewed for patterns and most used methods from the teachers. These patterns being used for several teachers meant that they were likely effective at helping students improve their comprehension and develop better problem solving skills [14]. The patterns discovered were closely related to variables roles [13]. PandionJ takes leverage of the previous illustrations and mimics them in its graphical interface to show it to the student [13].

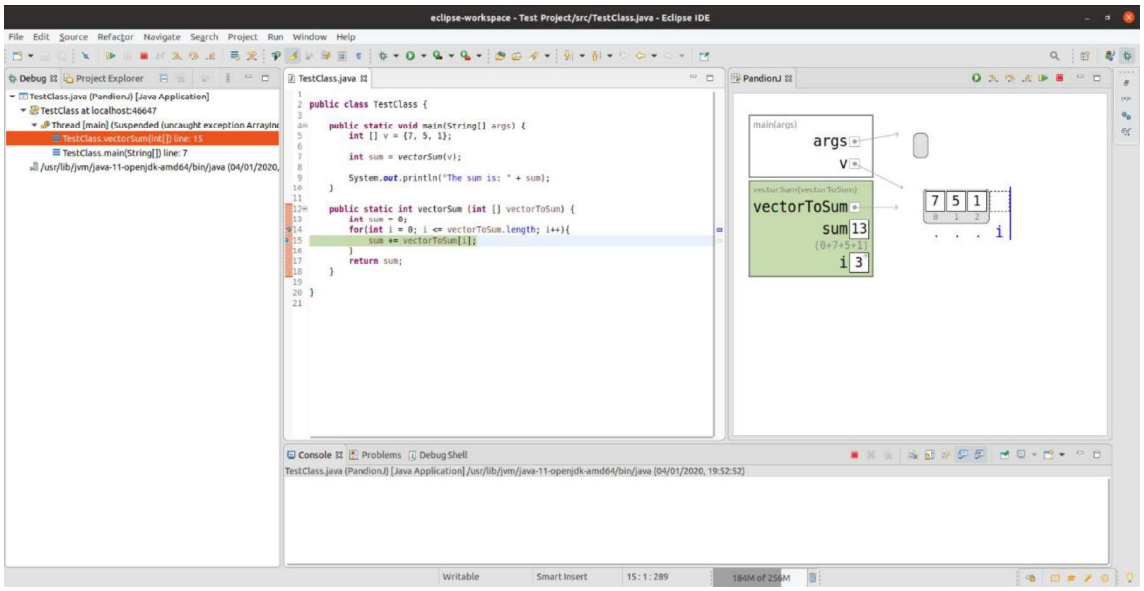

Figure 2.6 - PandionJ displaying the same example used in Figure 2.1

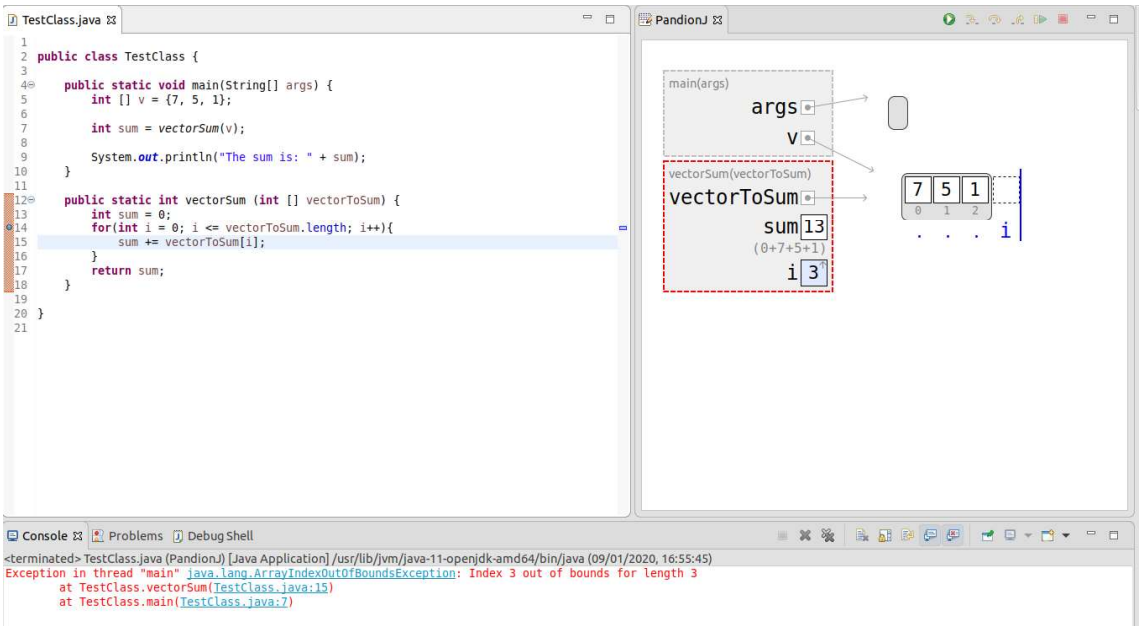

Figure 2.7 - PandionJ when the exception displayed in Figure 2.2 happens

Despite the debugger displaying runtime execution in a much more informative and simpler way for a student, it does not focus in execution errors. The object which had an error is highlighted, but no further explanation is given. That lack of information means that the student may be stumped when such error occurs and be unable to properly debug and understand what is happening. Such information may come from a proper explanation of how a specific execution error occurs and frequent reasons.

3. Enhancing explanation of Java execution errors

This chapter describes the work developed to achieve the first objective of this thesis. This was fulfilled with the development of a prototype to provide the student with extra information upon encountering an execution error during his early programming learning stages.

3.1. Illustrating Java Execution Errors

In order to improve a student's comprehension of execution errors, we must consider the human working memory characteristics. As described in section 2.1, providing a person with better explanations and illustrations of a certain topic can help her with achieving a greater understanding of the topic. With that knowledge in mind, this work aimed to provide the student with a good amount of information but without sacrificing simplicity, since a convoluted message could cause the student to become confused or simply not understand what is being transmitted.

For the purpose of this work and answering the research questions, this prototype only supports the ArrayIndexOutOfBoundsException, but the functionality could be extended to other exceptions. This type of exception is one the most common in beginner programming, especially when starting to learn these data structures, thus making it a good candidate for the purpose of this work.

Java's stack trace provides only the essential information of what happened and where but does not say why it happened, at least in an obvious way for a beginner, as seen in Figure 2.2. This means that one of the points the developed prototype would have to improve on is the explanation of the Exceptions. First, a textual explanation of what is happening. For example, an ArrayIndexOutOfBoundsException can simply be explained as "An error that occurs when iterating outside of the array's valid indexes", but simply providing this as an explanation is not enough to contextualize the problem that originates an ArrayIndexOutOfBoundsException. Extra information such as what is the size of the array, the variable that represents the array (useful in cases with multiple arrays in the same line of code) and the variables that were used to access that same array could help a novice better understand what is happening and where to fix the problem.

Building upon the textual description should be a graphical illustration of the error. A beginner requires more information in order to understand a problem, in which a text may not be enough to clarify his doubts or even to capture his attention to the problem. Providing context in a graphical way boosts the readability of the information, by complementing the information already given by the text description using a visual language that provides more possibilities of description than text.

Furthermore, the illustration can even offer additional information over what is already given by the text.

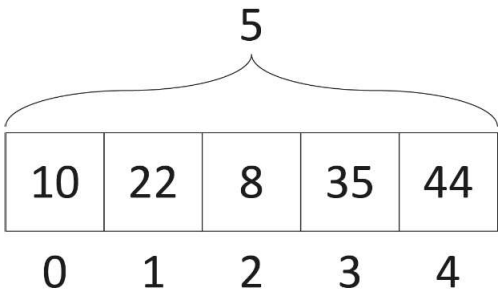

Figure 3.1 - Early Draft of an array illustration

One of the elements of the visual representation that had a special attention were arrays. These data structures can be complicated for beginners to grasp and understand what happens inside them, so a correct and clear illustration is critical for their understanding of how they work. Displaying the array the user is interacting with in a graphical way can help understanding everything easier. It allows for better tracking of what happened during execution and an easier way for the student to independently discover what went wrong in its approach. Figure 3.1 features an early draft that tried to fulfil all these points and transmit an efficient message to the end user.

3.2. Roles of Variables

One of the elements which could help beginners better learn how to use and interpret variables are roles of variables. These were explained in 2.2 and in this section they will be complemented with their interpretation and implementation in the prototype. From the list previously mentioned in Table 2.1, only two were implemented in the context of this work, which are the fixed value and the stepper, with an extra one being the array index iterator, which is a sub-role derived from the stepper. Moreover, the roles only focus on the detection of primitive values and arrays of primitive values, which means that objects are not supported and will not be a part of how the roles work and behave, since a student learning Java starts with these basic data types and only advances into objects further down the line.

3.2.1. Fixed value

A fixed value is characterized as a variable that does not change its value after being assigned. In the context of a method, the variable can either be received as a parameter, and therefore already have a value, or it can be created and assigned inside of the method. If the variable is a parameter, it

is considered a fixed value if its value is never changed during the execution of the code. On contrary, if the variable is not a parameter and is created inside of the method body, it is considered a fixed value if after the first assignment its value is never changed. This includes variables in which their first value is given by another variable or an expression with several variables and not altered further down the line during execution of the code. In the case of arrays, the previous applies with a few extra differences. An array is considered a fixed value until it is created again, exactly like normal variables described above. Adding to this is the possibility of switching a value inside one of the array positions. To deal with that, the fixed value can have a modified attribute, specific to arrays, which indicates if one of their internal values has been changed. This means that an array can be a fixed value that has been modified, to distinguish them from a fixed value array that is not modified internally. Figure 3.2 demonstrates the entire process through a diagram, for easier comprehension.

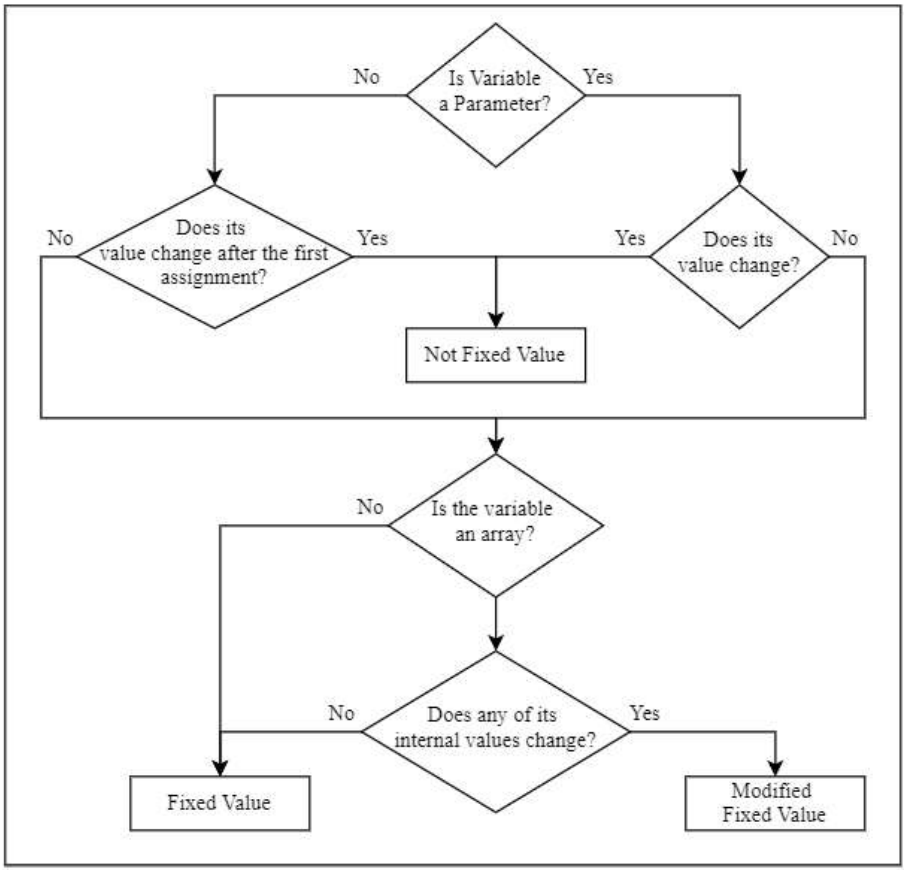

Figure 3.2 - Fixed Value Verification Diagram

3.2.2. Stepper

The stepper is another of the existing roles of variables described in Section 2.2. Compared to fixed value, this stepper implementation requires more checks in order to verify if a variable is indeed a stepper or not. A stepper is a variable in which its value goes through a predictable succession of

values. The first requirement is that the expression that gives the variable its values must be a binary expression. A binary expression is an expression which contains two operands separated by an operator (for example, i + 1). One of these operands must be the variable itself, for the variable's next value to be the previous value added with another value. In order to maintain the predictability, the other operand must be a literal, also known as a constant, so each incrementation is of the same size. Another requirement is the binary expression must be either an addition or a subtraction. Typically, these two operators are the most used for a stepper, and in order to simplify both the analysis and the comprehension of the role, only these two are allowed for the stepper role. When the operator is a subtraction, there is one extra verification that must be done, which is that the variable must be on the left side of the binary expression and the literal on the right side. If the literal is on the left side, there is no continuation of the previous value, which is a requirement listed above, due to subtraction, contrary to addition, not having the associative property. This property determines that when present the order of the elements of an expression does not matter, and the result will always be the same. Addition does not have any extra specific requirements in its validation and order is not important. The stepper also has an attribute that indicates which direction it is going, positive or negative. For example, if it is a sum of two positive numbers, the direction is positive, because the sum of two positive numbers is always a larger number. As another example, if it is a subtraction of two positive numbers, then the direction is negative. This entire process is shown in Figure 3.3.

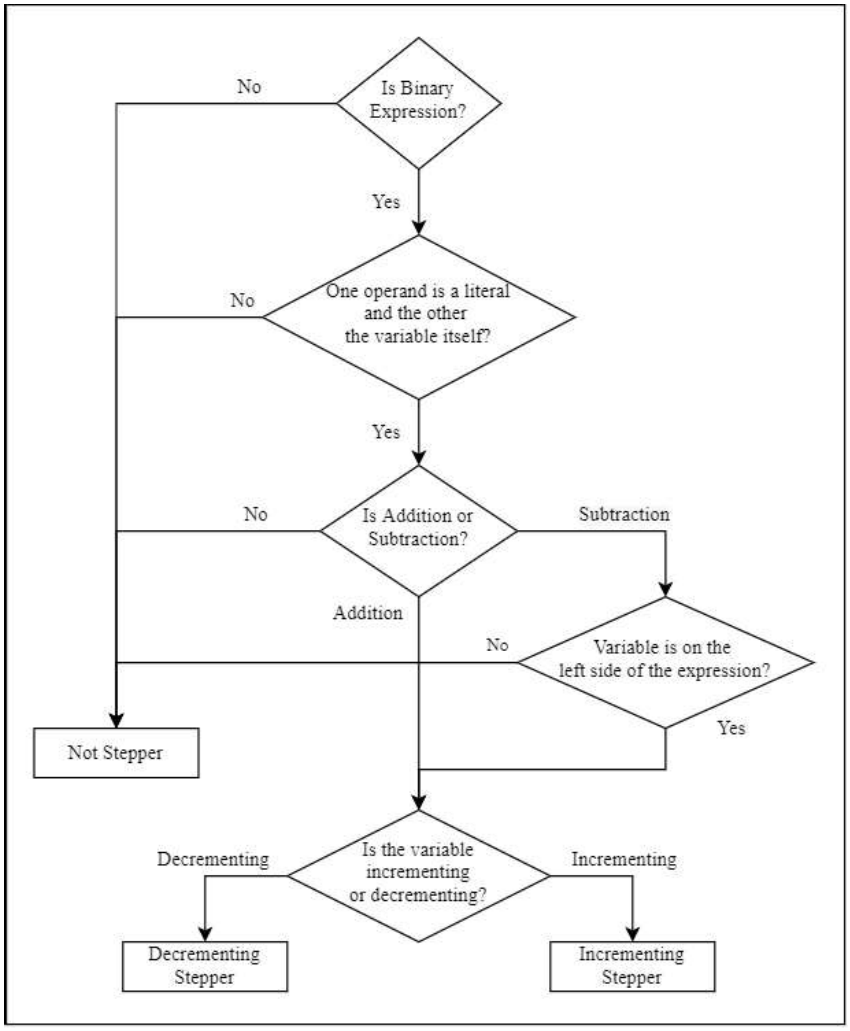

Figure 3.3 - Stepper Verification Diagram

3.2.3. Array Index Iterator

The array index iterator is a special case of a stepper, hence why it is not included in Table 2.1. This role applies to when a stepper is used to access a position of an array. Before starting to check if the variable is an array index iterator, it is required to validate if it is a stepper. After this, the only verification required is to check if the variable was used to access an array. The array index iterator can also store which arrays it accessed for later consultation.

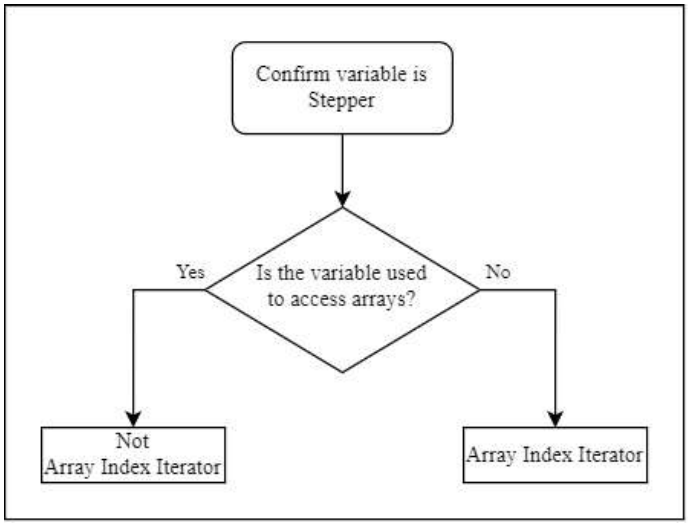

Figure 3.4 - Array Index Iterator Verification Diagram

Figure 3.4 details the rather simple process to determine whether a variable is an array index iterator after confirmation it is a stepper. A variable being a stepper is completely independent of it being an array index iterator, which means it can be a stepper and not be an array index iterator, but the other way around is not possible.

3.3. Graphical Interface

The prototype was designed with the main objective of being simple but informative. In order to fulfil this objective, the design of the interface would also have to follow the same principles. The user interface was written in Portuguese, which means this section's figures of the prototype's interface will contain text written in the latter. The final product's graphical interface was built to accommodate these requirements, culminating in what can be observed in Figure 3.5 and Figure 3.6.

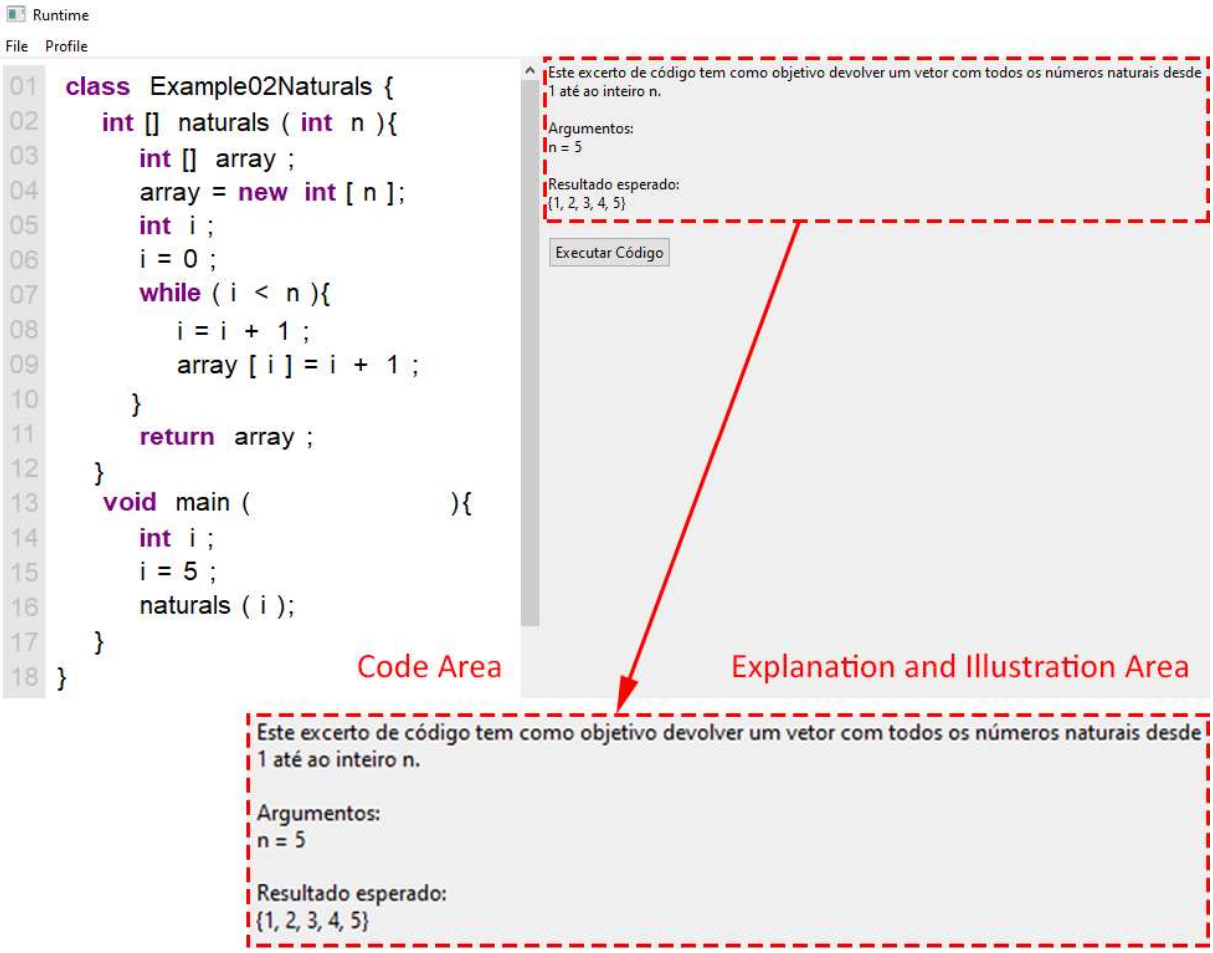

Figure 3.5 - Prototype's graphical interface

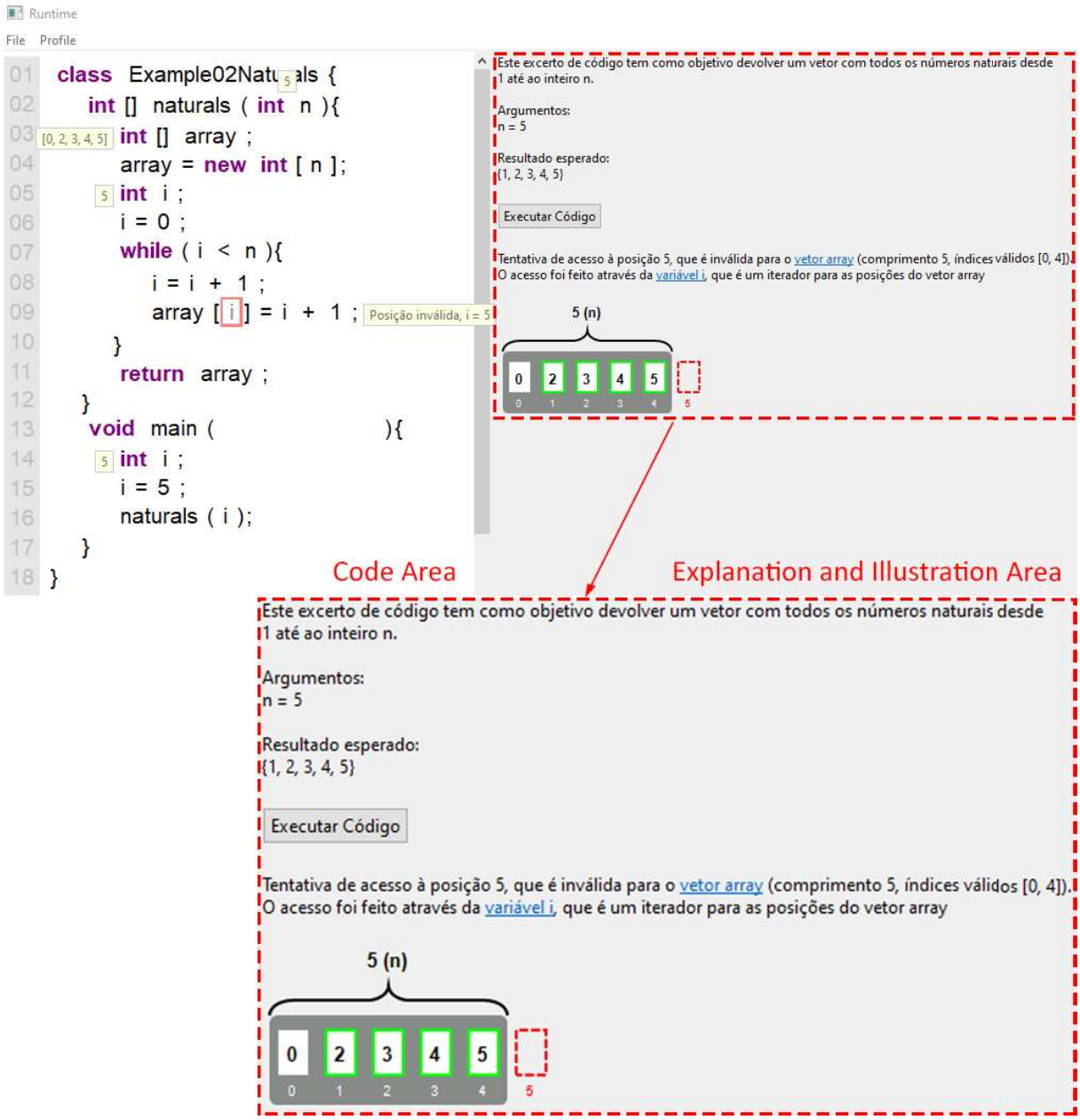

Figure 3.6 - Prototype's graphical interface after executing code

The program's window can be divided in 2 main areas. The left portion is the code area, where the code is located and displayed, as would be in a normal IDE. It supports some features such as a simple syntax highlighting and line numbers but also includes some details specific to this work. The right portion is where most of the major focus of the prototype is done. This last area is the focus of this work and is the visual representation of what was described as intended in the prototype in section 3.1.

Figure 3.6 contains a full demonstration of everything the prototype offers the end user. In the code area, extra annotations pop up and provide information of what the state of each variable is when execution of the program stops due to the execution error. First is the red square, which shows the expression that causes the error and to the right of it an annotation that says what was the problem

and the variable's value, in this case it was an invalid vector position when i equals 5. In this example the vector was accessed using a single variable, but the red square highlight supports multiple variables in case the access is done via an operation like a sum of variables, for example. In addition to the already mentioned annotations, each of the variables created inside the method are also annotated in the left with the value they had when the code threw an exception, as seen in line 5 applied to integer i, and the same applies to each of the method's parameters, as seen in line 2 which shows the value of the integer n.

Figure 3.7 - Explanation and Illustration Area from Figure 3.6

Moving on to the Explanation and Illustration area of the prototype, it can be divided into two main sections, one is the code contextualization and the other is the code explanation. The first section contains information about what the displayed code is supposed to do and return upon success. This is shown as soon as the code is loaded and appears before the code is executed. As seen in Figure 3.7, this is displayed in the top part of the explanation and illustration area and is consisted by 3 pieces of information, labelled one to three. The first one is a description of the objective of the method displayed in the code area, in this case returning the sum of all integers inside an array. The second part is an example of a possible argument that can be passed to the function, which in this example is the value 5. The third and final part is the expected return value of the method when the argument in the second part is given and if the code did not produce an exception, in this case an array of size 5

with the values “1, 2, 3, 4, 5”. These pieces of information provide the context for the student to understand what the pre-existent exercises try to achieve.

The second section is where the prototype provides more information to the user related to the objective of the work. As written previously, the code area gives some insight about the state of the variables when the code throws an exception, and this new area delivers the rest of the information related to the execution of the code. For starters, the button located on top serves the purpose of running the code and seeing what is returned. Upon running the code, if an exception is encountered, all the extra information related to the code’s execution is calculated and provided to the user. This means that the code area’s annotations, the code textual explanation and the array illustration are all created at this moment. Below this button is the code textual explanation, which gives the user a textual explanation of the problem encountered in the code. This part aims to provide the student with a global description of exactly what happened. Information such as array size and valid positions (to remind the user that an array of size 5, for example, has numbers one to four as valid indexes), invalid index that the code tried to access, description of the illegal action, variables that contributed to the invalid access and if possible their respective variable.

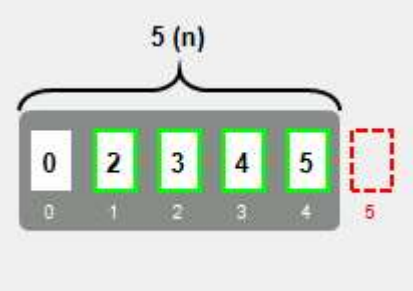

Figure 3.8 - Array Illustration Example from Figure 3.7

The final part of this section is the illustration of the affected array or matrix. This area displays the array in a graphical and informative way and helps the user develop a mental image of how an array is structured, which will help in the long run in the development of beginner knowledge. Figure 3.8 shows in a better highlight the illustration of a “normal” array, also known as a unidimensional array. The darker grey rectangle represents the array’s structure, with everything inside it being the elements of the array, represented by the small white rectangles. Inside each of these smaller rectangles is the value contained in that position of the array, while the smaller number below the rectangle represents the value of the index. Each of these rectangles can have a green outline, which indicates that during the execution of the code at some point that position of the array was accessed, either for reading or for writing data. Not having the green outline means that the position was never accessed during the execution of the code. Moving on to outside of the array, the red rectangle represented in Figure 3.8 on the right represents the index that was invalid. Since the array is in order, if this rectangle is in the right of the array, it means that invalid index is bigger than the biggest valid

index, and if the rectangle is in the left, it means it's a negative number, since only negative numbers are lower than the lowest array index, 0. The number below the red rectangle is the invalid position that the code tried to access. Above the array is a representation of its size, outside of the parenthesis, and inside of them is the expression that calculates array's size. This expression can only be obtained if the array is created inside of the analysed method, because information related to that expression cannot be obtained from inside of the method if the array is received as a parameter.

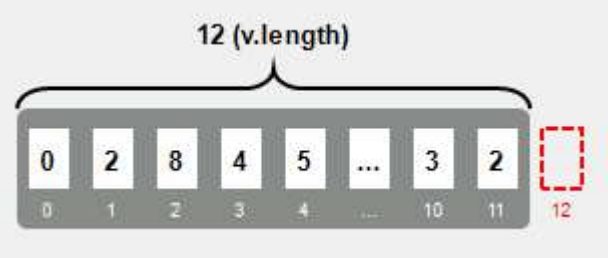

Figure 3.9 - Shortened Array Example

This illustration has a limited amount of horizontal space. If an array is big enough, trying to draw it on screen would cause the illustration to go out of bounds and become unreadable. To circumvent this problem, the array is shortened to a maximum of eight values displayed at a given time. If an array has a length of eight or less, all values can be displayed normally, as already demonstrated in Figure 3.7 and Figure 3.8. When the length is bigger than eight, the first five and the last two elements are shown and all intermediate values are omitted, as shown in Figure 3.9. This allows for a reasonable amount of information about the array contents while keeping the illustration size manageable and readable.

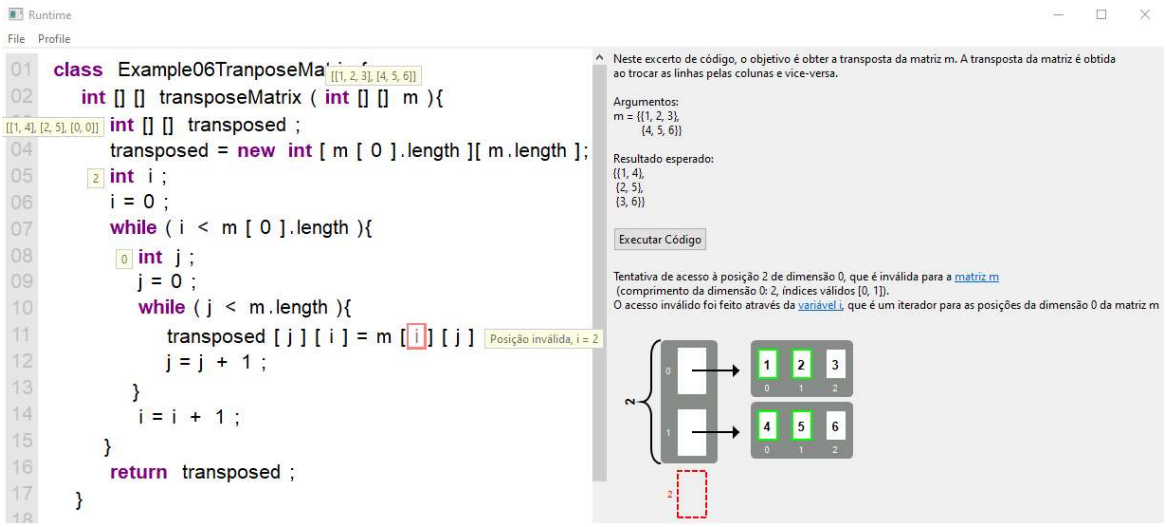

Figure 3.10 - Prototype's interface with a method involving matrices

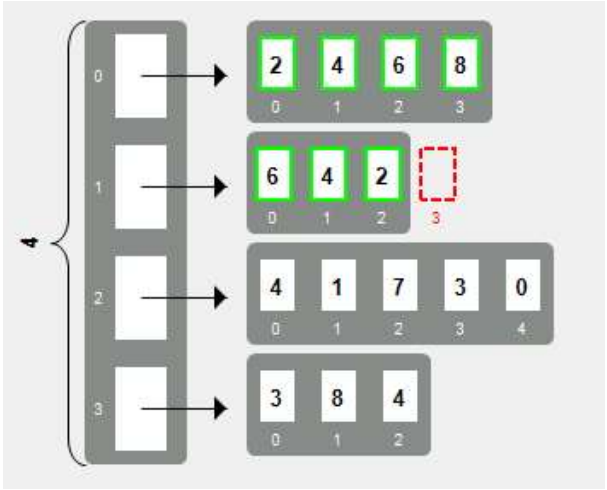

Figure 3.11 - Matrix illustration example with different row sizes and horizontal array error

When the method involves a matrix, the previous illustration does not fit well with what is required to represent it. Since a matrix is a two-dimensional array, the logic used for one-dimensional arrays can be used with a couple of upgrades. One of the points this illustration tries to convey to the users is that a matrix in java is an “array of arrays”, which means it’s an array where every position is itself another array. As seen in Figure 3.10, this is achieved by drawing an array on the left side in a vertical position, where instead of each of its elements being a value, they have an arrow which points to another array where that one is the one that contains the values. Invalid positions can appear either on the vertical array or in the horizontal arrays and different sized horizontal arrays are supported, as shown in Figure 3.11, just like in regular java where rows may not all be the same size. The rest of the array symbols apply apart from horizontal arrays not having the array size written, as such would occupy too much space and cause a lot of visual clutter. Like regular arrays, matrices can also be shortened if they are too big to fully illustrate in the available space. Both vertical and horizontal arrays behave similarly to Figure 3.9, with the same constraint in maximum displayed array elements.

3.4. Architecture

This section will feature an overview of the process and artefacts that are used to provide the described tool features of the prototype and its inner workings and elaborate about details related to the logic behind the implementation of all the features included in the prototype and how they work.

The tool addresses exceptions, which only happen during runtime. Simply performing a static analysis of the code does not allow to obtain information about exceptions since these are not apparent and cannot be obtained via this method. On the contrary, variable roles use a static code

analysis, and are not obtained in the same way as the exceptions. This means that the prototype will have to feature both types of analysis in order to determine every piece of information necessary for generating both the textual explanation and the illustration.

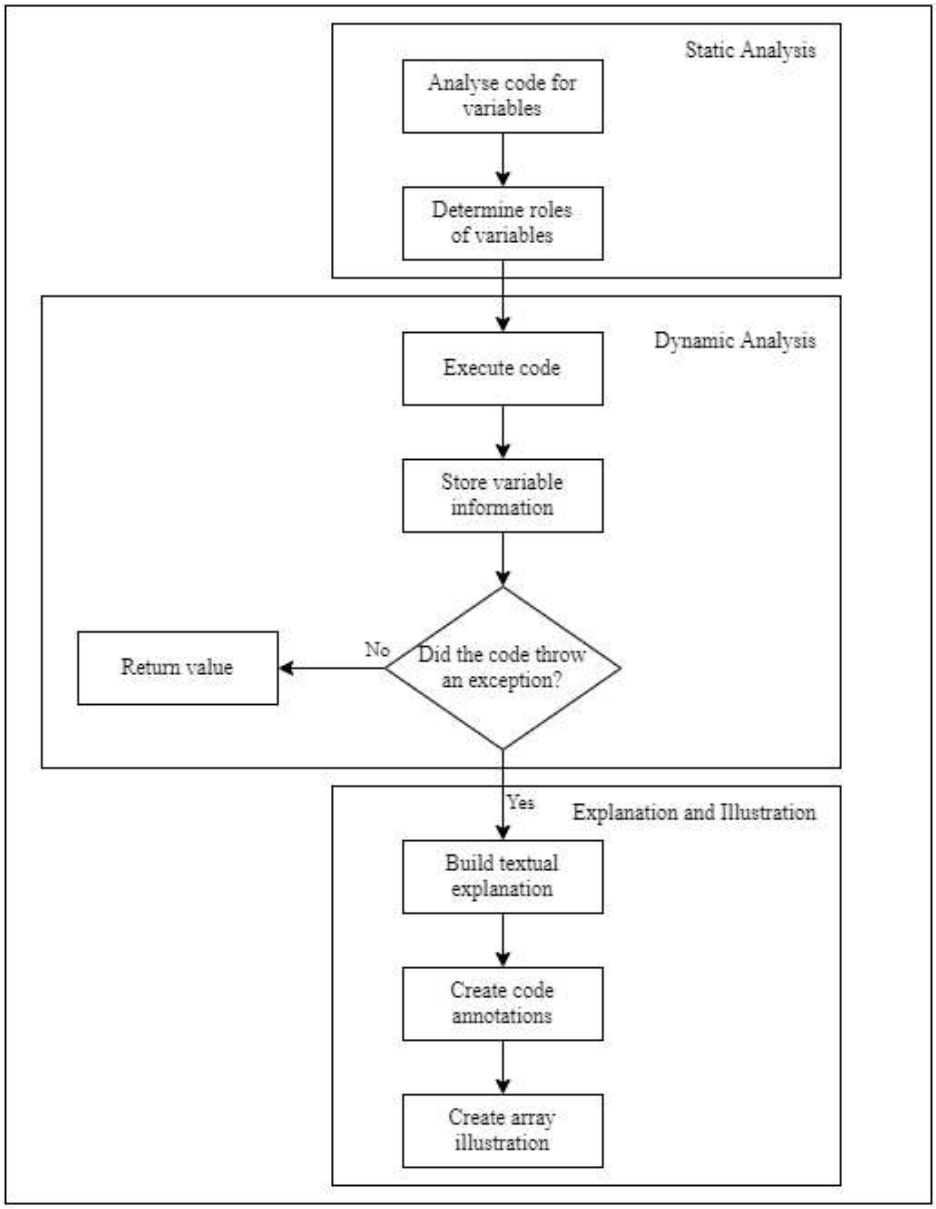

Figure 3.12 - Code analysis process diagram

3.4.1. Static Analysis

The first step in analysing a snippet of code is performing a static analysis. This is the shortest part of the whole process and its purpose is to obtain the roles of each variable. It starts by sweeping the code to obtain all existing variables. After that, each discovered variable is analysed against each of the existing roles, each one with its own distinct process, to determine which one fits the use case. The roles and each of their requirements are explained in section 3.2. They are tested in a specific order,

starting with fixed value, then stepper and finally array index iterator. The fixed value is tested first since it is simpler than a stepper to test. If the variable is a stepper, it must also be tested for an array index iterator. If the latter is also confirmed, the variable is an array index iterator, if not, the variable is a stepper. It is possible for a variable to not fit the requirements of any variable role, especially since only three roles are being used in this prototype.

3.4.2. Dynamic Analysis

Running a piece of code in the prototype is a multistage process from the beginning to the drawing of each illustration. Pressing the code executing button in the interface starts the process and despite the prototype containing the code examples prewritten, the process is entirely dynamic and can be used with any piece of code within certain limitations, such as no support for objects, which means all variables must be primitive data types. This analysis is possible thanks to an execution engine[15] that can execute provided code and allow access to the stack trace and execution state. The engine also offers support for listeners to be registered and listen for specific events that happen during runtime.

The runtime engine starts the execution of code in a chosen method, that will act as the main method for execution, but can have any kind of return type and any name. When the execution starts, the first thing that is done is checking the method's parameters for their initial values and store them appropriately. After that, a listener is created to listen to what happens during execution of the method's code. This listener is triggered for every statement of the code and the variables contained in it and their respective values are stored for future use in evaluating what happened during execution. A variable can be accessed multiple times and its value can change throughout the execution of the code. Because of this fact, the prototype stores the history of all values assigned to each variable in order, so that it maintains the complete information about variables history.

For each existing variable, three pieces of information are stored for further use: variable type, history of all values and a reference to the variable. The variable type can be two different possibilities, either the variable is a parameter or a local variable. The variable history explained before benefits from the added simplicity of only dealing with primitive data types, which are easily stored in a list that sorts by order of insertion. The last part is the reference to the original variable, which is useful internally and since this is all done inside of the same runtime environment, it is possible to access even after the code stops executing. This information applies to all variables that can be encountered, but arrays are a special case which require extra attention.

Arrays are more complex than a normal variable and extra information is required, which means that extra data must be stored for analysis further down the line. An important note regarding arrays

in this prototype as well as Java in general is that a matrix is an array of arrays (i.e., an array in which each position holds a reference to another array). This means that the information for unidimensional arrays is enough for arrays with any dimension, making this approach viable for both normal arrays and matrices, which are both considered in the execution process. Therefore, regarding the collection of variable information, any further explanation will work for both arrays and matrices unless explicitly stated otherwise. For arrays, apart from the information collected for all variables, three extra pieces of data are obtained. For starters, the number of dimensions, since this is required to distinguish an array from a matrix, which is essential. The second is the history of all array accesses, which can be stored in a list with the indexes of each access as values. The final piece of information is the length expressions that created the array. These are the expressions that were provided for each dimension when creating an array, not the size itself, and are optional because when an array is received as a parameter this is not possible to obtain.

The execution of the code and collection of data continues until one of two things happen: either the code finishes, and an end value is returned, or an execution error is detected and execution halts. Since the normal functioning of the code does not belong to the scope of the project, when such event happens the only thing returned is the correct value of the code's execution. On the other hand, an exception occurring triggers the continuation of the process, which will be then move on to the next part of the process.

3.4.3. Explanation and Illustration

To start the next part of the process, the first objective is checking which type of exception has occurred. The runtime engine can detect what type of exception was thrown by the code, which then allows the prototype to act according to the requirements of the exception. As already explained in Section 3.1, only ArrayIndexOutOfBoundsExceptions are being analysed in this thesis, which means despite other types of exceptions being possible, only this type is supported by the prototype. In Section 3.3, it is shown what the textual explanation and the illustration look like and what information they contain for the user. These are constructed in this part of the process, with the information gathered before.

After the exception is discovered, all the gathered information starts being processed in order to produce a text. One dimensional arrays and matrices have some text differences due to the number of dimensions involved, which means they require different text templates. Variable name and invalid access value are obtained the same way for both, with the only change being the invalid value changing the format in which it is written, because of multiple dimensions, but it is stored and accessed in the

same way despite that. The variable name is obtained from the reference of the array where the invalid access occurred. The invalid access value is stored in the history of access values specific to the affected array, more specifically the last access that occurred. The array size is done differently depending on the case. For arrays, accessing the array and obtaining its length is enough to write about its size and valid indexes (between zero and length minus one). In the case of a matrix, the text shows the size of the dimension where the illegal access occurred, which means the dimension must be obtained and written next to its size in the text, with the rest of it applying the format of a regular array. The last part of the text is the name of the variable that triggered the illegal access, which is easy to obtain thanks to the execution engine that allows easy access to variables that trigger exceptions.

After creating the text, the prototype generates the code annotations. These annotations are created directly in the code and provide quick and easy access to information about the variables they are next to. The most important annotation created is the red rectangle that highlights the variable or expression (if multiple variables are involved) that caused the illegal access and triggered the exception. Like the text, the execution allows easy access to this variable, so highlighting it in the code is just a case of searching for where it is used and drawing a rectangle around it. At the end of the line, an annotation is added that contains a small indication of the type of error that occurred (i.e. for an ArrayIndexOutOfBoundsException, it indicates “invalid position”), and is directly related to the exception, and also the variable that did the illegal access and its value. Finally, for every variable present, either a normal variable, array or matrix, the value assigned to it when the execution was triggered is annotated next to the variable for easy reference.

The last step is creating the array illustration. This illustration is the most complicated part of the last section of the process. For starters, the array contents are required to draw the values of each position. An array is illustrated as a grey rectangle with white squares inside representing each position, with the text inside them being the content of the position and below the square the index of that position. The array illustration is adaptable in size, which means the longer the length, the larger the horizontal size of the array. In order to be consistent, each position is the same size, which means an array of length four is roughly double the size of an array of length two. The counterpart is that space is not infinite, so a limit had to be imposed in order to not exceed the graphical interface’s limit. That limit was defined as eight array positions. If an array is length eight or lower, the prototype simply draws an array of that size with each position having its respective value. If the length is greater than eight, the prototype draws an array of that length and copies from the full array the beginning five values (index zero to four) and the last two (index array length minus two and index array length minus one). This leaves the sixth position of the illustration free, so the prototype inserts a symbol to indicate it has been truncated (“…”). After drawing the array itself, the top part where the array length and

expression that provided the length value are created. The length expression is optional, as already stated in section 3.4.2, so it is only drawn when possible.

When drawing matrices, the logic is slightly changed but most of the unidimensional array code can be repurposed. Since the prototype was directed at Java beginners, the matrix illustration focuses in certain core ideas of how they work. A matrix is an array where each position is another array, and this illustration is designed to transmit this information. For this purpose, a vertical array is drawn, similar to the previously explained arrays, but in which each position instead of containing a value, they have an arrow point to another array which represents that “line” of the matrix. The vertical array also has an array size representation and length expression which are related to the first dimension of the matrix. The horizontal arrays do not have the size indication because it would occupy to much space and the length expressions may not make sense because a matrix can have lines of different lengths.

4. User Study

In this chapter, the procedures and results of the study are described. The experiment was conducted to fulfil the second objective of this thesis and provide an answer to the second research question. The purpose of this is to validate whether the developed pedagogical tool had a positive effect on students' comprehension of execution errors and how to avoid them.

4.1. Method

The prototype is a pedagogical tool focused for people that do not have knowledge about execution errors, therefore hitting a wall when they face such problems. Following this idea, the recruitment stage was planned to target an audience of students that were in a level where such adversities occur. The first requirement for recruitment candidates is being a student at Iscte. Being from the same university means that all possible candidates that are on courses with programming subjects were all taught the same contents. The second criterion is the course the students are attending. Arrays and matrices are learned in introduction to programming in Iscte, so the preferred level of learning was students finishing introduction to programming or starting object-oriented programming (second programming subject). These two requirements allowed for students who understood how data structures such as arrays worked but may still have trouble dealing with execution errors, knowledge that is further developed during the second course. Students were contacted for recruitment via email. The lists of students that were attending introduction to programming and object-oriented programming were used to contact potential candidates.

The experiment consisted in an interview with each participant in which they had to answer questions related to six different methods. The six different methods are included in Appendix A. They consist of faulty code that will face an execution error when ran. The code would then be executed and return an execution error in one of two ways: either the normal stack trace, or with the explanations and illustrations written in this work.

We carried out a within-subjects experiment, especially due to the low number of available subjects. The six methods were divided into two different groups, one for odd numbered methods and another for even numbered. The objective of separating these two groups is comparing the differences between the two alternatives in similar coding cases. The candidates were also divided in two different groups, half for each. For the first half, the odd numbered methods, when executed, would return a normal stack trace (Java's output) and the even numbered methods would return the illustrations of our prototype. The other half of the participants had the inverse sequence of error output.

Each method's execution was followed by a few questions related to the code's errors and possible corrections. The first two questions were about locating the error in the code and trying to explain why it happened. These were open questions to which participants answered orally. The next question was a multiple-choice question where a section of the code that caused the execution error was highlighted and each answer was a possible replacement for that area. The final question, also a multiple-choice question, was not directly related to the method but was a more general question to evaluate the students' knowledge of basic concepts related to arrays and matrices. Both multiple choice questions could contain multiple correct answers or even no correct answers. All questions were repeated for each of the methods and are included in Appendix B.

To better explain how the experiment unfolded, the procedure for one of the methods will be described in this paragraph. This procedure is repeated for each of the methods, so by describing one of them the logic applies to the rest. For starters, the interviewer shows the code to the volunteer and gives a brief explanation, running the code in the process. The only change this step has is whether the question shows the java stack trace (Figure 4.1) or the explanation and illustrations (e.g., Figure 3.6). After giving a brief explanation, the interviewer waits for the participant to answer the open questions and writes them down. After answering, the interviewer changes to the first multiple-choice question and waits for the participant to say the correct options. Finally, the interviewer moves on to the last multiple-choice question, and proceeds exactly as the first multiple-choice question.

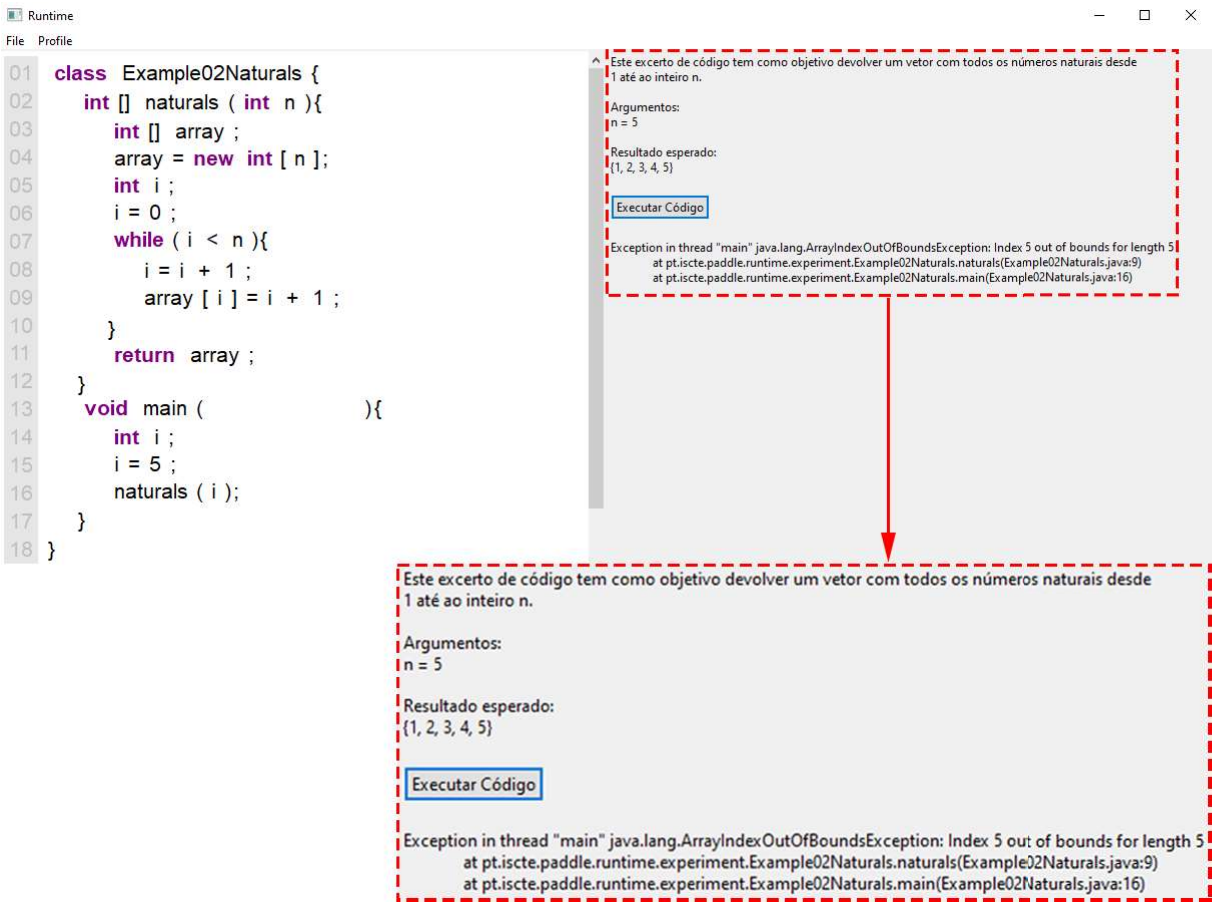

Figure 4.1 - Interface when showing regular Java stack trace

4.2. Results and analysis

The volunteers that we managed to recruit are students from the computer engineering and telecommunications and computer engineering degrees offered at our institution, who have obtained approval on the subjects of Object-Oriented Programming (second programming subject) or Concurrent and Distributed Programming (third programming subject). Despite the best efforts for reaching volunteers through email advertisements sent by the course professors, only a total of six volunteers accepted to participate in the experience, resulting in a small sample to observe.

As written in section 4.1, the participants were separated into two groups, three volunteers for each. Group A was the group that had odd numbered methods using the regular stack trace, whereas group B had the opposite with even numbered methods using illustrations.

Table 4.1 - Results for the open questions

	Group A				Group B			
	Correct	Wrong	No answer	Score	Correct	Wrong	No answer	Score
Method 1	3	0	0	3	3	0	0	3
Method 2	3	0	0	3	3	0	0	3
Method 3	3	0	0	3	3	0	0	3
Method 4	3	0	0	3	2	0	1	1
Method 5	3	0	0	3	3	0	0	3
Method 6	3	0	0	3	2	0	1	1

Table 4.1 displays the results for the open questions, where the volunteers must locate the error and explain why it happens. The numbers in the correct, wrong and no answer columns refer to the number of volunteers who had this classification in that method. The score column is a sum of the results from the other columns, for easier observation of the overall performance of the volunteers. Grey rectangles highlight the cases of the experimental group that used our tool. The criteria for classification was as follows:

- Correct: the participant located the error in the code and explained why it happens – increases score by one
- Wrong: the participant located the error and explained but incorrectly – decreases score by one
- No answer: the participant did not manage to locate the error or explain why it happens - decreases score by one

In Table 4.1, it is possible to observe that almost every question was answered successfully. Group A had 100% correct answer rate and does not allow for any conclusion alone. In Group B however, in two different methods, there was one volunteer that did not know how to answer the questions, with both cases being a method which only used the regular Java stack trace and not the illustration tool. This may point to the possibility that the tool offers better information for the user to be able to detect execution errors, although the difference between answered and non-answered questions in the table is insufficient to be sure.

Table 4.2 - Results for code multiple-choice question

	Group A				Group B			
	Correct	Partially Correct	Wrong	Score	Correct	Partially Correct	Wrong	Score
Method 1	3	0	0	3	3	0	0	3
Method 2	3	0	0	3	3	0	0	3
Method 3	3	0	0	3	3	0	0	3
Method 4	3	0	0	3	2	0	1	1
Method 5	3	0	0	3	3	0	0	3
Method 6	0	0	3	-3	1	0	2	-1

Table 4.2 shows the results for the first multiple-choice question, in which the participants had to choose possible code solutions. The columns for this table were slightly different, but still represent the number of students that had that classification in each method, with the score table being the sum of the results. The criteria is slightly different from before to better fit the evaluation of multiple-choice question, and was made as follows:

- Correct: all answers are chosen, or if none is correct, the participant said so explicitly – increases score by one
- Partially correct: only some of the correct answers are selected, and no incorrect answers are chosen – adds zero to the score, or in other words, score is unchanged
- Wrong: At least one wrong answer is chosen, even if the rest are correct – decreases score by one

Group A was correct in all answers except in the last method, in which everyone failed, despite being a method which contained the explanation and illustrations. Group B was similar except in method four and six. In method four, which only had the stack trace, one person failed to correct the code, but the rest succeeded. In method six, this time with the stack trace, only one person managed to answer correctly, while the rest failed. The last question being almost universally wrong may have a relation with the fact that it was the only method where this multiple-choice question had none of the options as a correct answer, despite the volunteers being warned about this possibility. In method four, group B which had the stack trace had one person incorrectly answer while the other group was completely correct. In the rest of the questions, the opposite happens, that is, all answers were correct.

Table 4.3 - Results for the general content multiple-choice question

	Group A				Group B			
	Correct	Partially Correct	Wrong	Score	Correct	Partially Correct	Wrong	Score
Method 1	2	0	1	1	2	0	1	1
Method 2	3	0	0	3	3	0	0	3
Method 3	2	1	0	2	3	0	0	3
Method 4	2	1	0	2	2	1	0	2
Method 5	1	0	2	-1	2	0	1	1
Method 6	2	1	0	2	1	1	1	0

Table 4.3 contains the results of the last multiple-choice question, in which the volunteers were evaluated with questions not directly related to the method, but similar. In this question, as opposed to the previous question, the results between the two groups differ, which hints that the tool might have had a positive effect. Methods one, two and four had equal results across both groups, despite not all of them being completely correct. In method three, group B had a better result, because A had one person with only a partially correct answer. Since this method is an odd number, that means the group with illustrations had a better performance than the other. Methods five and six had a similar outcome, where for method five group B had a better result, with only one wrong answer versus the two from group A and method six being the inverse and having group A with better answers than group B. This means that whenever groups scores differ, the experimental group using our tool gets the highest score. From analysing Table 4.2 and Table 4.3, some conclusions can be drawn. In the first table, we can observe that the tool made no significant difference in the participants' capability of fixing the code. Since most answers were correct, there is a strong possibility that the questions were too easy for the level of learning the students have, which may be the reason for having no noticeable differences. When analysing the second table, the same does not apply. In this, there is a noticeable difference between the correct answer rate of both groups, which points to the possibility that the tool can help with the students' capability of answering questions with similar topics. This may be related to improving interpretation of future questions with the same topic when a better explanation is provided. The low number of participants does not allow for statistical significance, but the results encourage further experiments to confirm if there is such an effect.

Table 4.4 - Average time necessary to answer questions (minutes:seconds)

		Group A	Group B	Fastest Group
Method 1	Open questions	0:52	0:46	B
	Code multiple-choice	0:14	0:18	A
	General multiple-choice	0:14	0:17	A
Method 2	Open questions	0:55	1:47	A
	Code multiple-choice	0:21	0:35	A
	General multiple-choice	0:28	0:43	A
Method 3	Open questions	0:59	1:39	A
	Code multiple-choice	0:22	0:46	A
	General multiple-choice	0:21	0:33	A
Method 4	Open questions	1:31	2:15	A
	Code multiple-choice	0:49	1:21	A
	General multiple-choice	0:47	0:34	B
Method 5	Open questions	1:37	0:56	B
	Code multiple-choice	0:11	0:55	A
	General multiple-choice	0:18	0:28	A
Method 6	Open questions	1:51	2:18	A
	Code multiple-choice	0:55	1:12	A
	General multiple-choice	0:24	1:05	A

Table 4.4 displays the average amount of time the participants took in each question. The average is calculated with all 3 times in each group except in the open questions when a participant explicitly says he does not know the answer, not including that time in the calculation. Since both open questions are asked at the same time and the participant can answer both freely, they were grouped together as the same time. The last column contains the letter that represents the best group for that question. Group A can answer faster than group B almost every time, with few exceptions, which makes it not viable to make any conclusions related to the impact the tool has in the speed in which the users can answer.

One observation that happened during the experiment is related to students changing their answers. Although not a common occurrence, three of the interviewees had one question each where while they were answering verbally, they changed their mind and answered the question in a different way, correcting the answer. When this phenomenon happened, the sequence of events was similar for all students. They started by locating the error as the first step, following the order in which the open questions appear. After, when trying to answer why it happened, there was a waiting time where students were quiet or tried to answer but stopped mid-sentence and did not know how to proceed

with the answer. After a while, they would suddenly realise their line of thought did not work and changed their answer to something different, thus giving the correct answer. The most important part of this phenomenon is that it only occurred when the method was shown with the regular stack trace instead of the tool. This is particularly interesting because it may indicate that the tool helps students reach their answer in a more direct way, avoiding incorrect lines of thought altogether.

The participants also gave some feedback regarding some aspects of the tool and how it could be improved. One of the interviewees mentioned that the placement of the value of variables in the code area (Figure 3.6) can be confusing because it appears above the line where the execution error appears, conveying the idea that this value was the value in that location of the code and not the value when the code threw the exception, despite the latter being the intention.

4.3. Threats to validity

In this section, some possible threats to the validity of this study are listed.

Low number of participants. The reduced number of participants may not allow for an accurate representation of the potential average user of this tool. Some differences noted between the groups, such as group A answering faster than group B in almost all questions is a symptom of this.

Interviewer interference. When interacting with the volunteers, the interviewer may unintentionally interfere with the experience by explaining the exercises in a different way to some of the participants when attempting to explain how the process works. This may have an influence how the interviewees interpret the questions and react to them, changing the outcome of the answers.

5. Conclusions and Future Work

In this work, the author created a pedagogical tool with the purpose of helping students achieve a better comprehension of how execution errors work and how to avoid or correct them if required. The work began with researching past tools and approaches to start off the tool. Guided instruction was the idea behind the tool and the type of learning it tries to provide. Roles of variables are one of the additions that were implemented to try and help students. Previously existing tools were analysed and reviewed for inspiration, including PandionJ, already used in Iscte in Introduction to Programming.

The answer for the first research question exists in the tool itself. The tool provides explanations and illustrations much more detailed than the conventional Java stack trace. Through the use of a runtime engine, it is possible to listen to events that happen during code execution and access variable values and states. Using this data, the tool provides information to the user of about the execution error that occurred. The first piece of information given to the user is the text. The tool generates a text which provides info about the type of error (which for this thesis is always an ArrayIndexOutOfBoundsException), size of the array, variable that caused the illegal access, the illegal index it tried to access and variable role. The second piece of information is code annotations. The annotations appear in the code area and indicate the specific location where the error occurred and the value of each existing variable. The third and final piece is the illustration. This is the most elaborate part of the three and is a representation of the array which was illegally accessed. It includes the values of all array positions, an indication of which ones were accessed, the representation of the illegally accessed index, the size of the array and, if possible, the expression that gave the array its length when it was created. These three things provide an answer to the first research question.

For the second research question, an experiment was conducted in order to determine the effectiveness of the tool in student performance. For this purpose, an interview was conducted with a questionnaire included in which students had to answer some questions related to this topic and the results were than analysed to extract data about the impact the tool may have had. The low number of volunteers for this study did not help with acquiring the required data, which would improve in quality if a bigger number of participants appeared. Even so, we concluded that the tool may help students finding errors easier and more directly, helping them in future situation like the ones the tool intervened in previously. In the open questions, although almost everyone was able to answer everything successfully, the only two exceptions that happened did so when only the Java stack trace was available. This meant that having access to the tool may have benefited the interviewees when trying to understand what caused the error. The first multiple-choice question did not allow for any conclusions because, like the open questions, there was a lot of correct answers. The difference is that

in this one the incorrect or partially correct answers were scattered through both the Java stack trace and the tool assisted answers. The last multiple-choice question revealed a different story. In this one, there was a bigger difference between the groups and methods, benefiting the tool much more than the Java stack trace. This means that even if the tool may not benefit the students directly when giving information in a method, it helps them create better knowledge for similar questions that may appear in the future. When measuring the average time it took for the participants to answer the questions, it was observed that one of the groups is faster than the other in almost all questions, with or without the tool's assistance, which means a conclusion from this piece of data is not possible. It was also observed that, while answering the open questions, there was a bigger tendency for the participant to alter their answer while answering this question, while with the tool such did not occur. This indicates that the tool helps students find and understand the errors in a more direct way, facilitating those with difficulties in this topic.

For future work, the tool could be improved with adding new types of execution errors and performing a better and more in-depth experiment. Different errors than the one already implemented could prove to be even more beneficial in helping students with their difficulties dealing with execution errors. Another important feature to add is the ability to write code directly inside the tool or allow compatibility with already existing IDEs. This would allow students to understand the code they themselves write and would the most important addition in order to allow this application to be used in a subject such as Introduction to Programming.

References

[1] M. Kuittinen and J. Sajaniemi, "Teaching roles of variables in elementary programming courses," *SIGCSE Bull. (Association Comput. Mach. Spec. Interes. Gr. Comput. Sci. Educ.*, vol. 36, no. 3, pp. 57–61, 2004.

[2] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," *J. Manag. Inf. Syst.*, vol. 24, no. 3, pp. 45–77, Dec. 2007.

[3] P. A. Kirschener, J. Sweller, and R. E. Clark, "Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching," *Educ. Psychol.*, vol. 21, no. 41, pp. 75–86, 2006.

[4] G. A. Miller, "The magical number seven plus or minus two: some limits on our capacity for processing information.," *Psychol. Rev.*, vol. 63 2, pp. 81–97, 1956.

[5] L. R. PETERSON and M. J. PETERSON, "Short-term retention of individual verbal items," *J. Exp. Psychol.*, vol. 58, p. 193—198, 1959.

[6] J. Sweller, "Cognitive load during problem solving: Effects on learning," *Cogn. Sci.*, vol. 12, pp. 257–285, 1988.

[7] J. Sajaniemi, "An empirical analysis of roles of variables in novice-level procedural programs.," *Proc. IEEE 2002 Symp. Hum. Centric Comput. Lang. Environ.*, pp. 37–39, Jan. 2002.

[8] J. Sajaniemi and M. Kuittinen, "An Experiment on Using Roles of Variables in Teaching Introductory Programming," *Comput. Sci. Educ.*, vol. 15, no. 1, pp. 59–82, 2005.

[9] N. Pennington, "Empirical Studies of Programmers: Second Workshop," G. M. Olson, S. Sheppard, and E. Soloway, Eds. Norwood, NJ, USA: Ablex Publishing Corp., 1987, pp. 100–113.

[10] J. Sajaniemi and M. Kuittinen, "Visualizing roles of variables in program animation," *Inf. Vis.*, vol. 3, no. 3, pp. 137–153, 2004.

[11] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The BlueJ System and its Pedagogy," *Comput. Sci. Educ.*, vol. 13, no. 4, pp. 249–268, 2003.

[12] M. Kölling, "Using BlueJ to Introduce Programming," in *Reflections on the Teaching of Programming*, vol. 4821 LNCS, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 98–115.

[13] 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 Calling '18*, 2018, pp. 1–9.

[14] 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 Conference on Computing Education Research - Koli Calling '17*, 2017, pp. 173–177.

[15] "GitHub - andre-santos-pt/paddle." [Online]. Available: https://github.com/andre-santos-pt/paddle. [Accessed: 28-Oct-2020].

Appendix A. Methods created for the experiment

```
1. int sum (int [] v) {
2.      int i;
3.      i = 0;
4.      int sum;
5.      sum = 0;
6.      while(i <= v.length){
7.              sum = sum + v[i];
8.              i = i + 1;
9.      }
10.             return sum;
11.     }
```

Figure A.1 - First Method - Return sum of all numbers in array

```
1. int [] naturals (int n) {
2.      int [] array;
3.      array = new int[n];
4.      int i;
5.      i = 0;
6.      while(i < n){
7.              i = i + 1;
8.              array[i] = i + 1;
9.      }
10.             return array;
11.     }
```

Figure A.2 - Second method - Return an array with all the natural numbers up to n

```
1.  int lastOccurrence(int [] v, int n){
2.      int i;
3.      i = v.length - 1;
4.      while(i >= -1){
5.              if(v[i] == n){
6.                      return i;
7.              }
8.              i = i - 1;
9.      }
10.             return -1;
11.     }
```

Figure A.3 - Third method - Return index of the last occurrence of an integer in an array

```
1.  int [] invert (int [] v){
2.      int [] v2;
3.      v2 = new int[v.length];
4.      int i;
5.      i = v.length - 1;
6.      while(i >= 0){
7.              v2[v.length - i] = v[i];
8.              i = i - 1;
9.      }
10.             return v2;
11.     }
```

Figure A.4 - Fourth Method - Invert array

```
1.  void scaleMatrix (int [][] m, int n){
2.      int i;
3.      i = 0;
4.      while(i < m.length){
5.              int j;
6.              j = 0;
7.              while(j < m.length){
8.                      m[i][j] = m[i][j] * n;
9.                      j = j + 1;
10.                     }
11.                     i = i + 1;
12.             }
13.     }
```

Figure A.5 - Fifth Method - Scale a matrix by multiplying each number by n

```
1.  int [][] transposeMatrix (int [][] m){
2.      int [][] transposed;
3.      transposed = new int[m[0].length][m.length];
4.      int i;
5.      i = 0;
6.      while(i < m[0].length){
7.              int j;
8.              j = 0;
9.              while(j < m.length){
10.                             transposed[j][i] = m[i][j];
11.                             j = j + 1;
12.                     }
13.                     i = i + 1;
14.             }
15.             return transposed;
16.     }
```

Figure A.6 - Sixth Method - Return a transposed matrix by swapping lines with columns and vice versa

Appendix B. Experiment questions

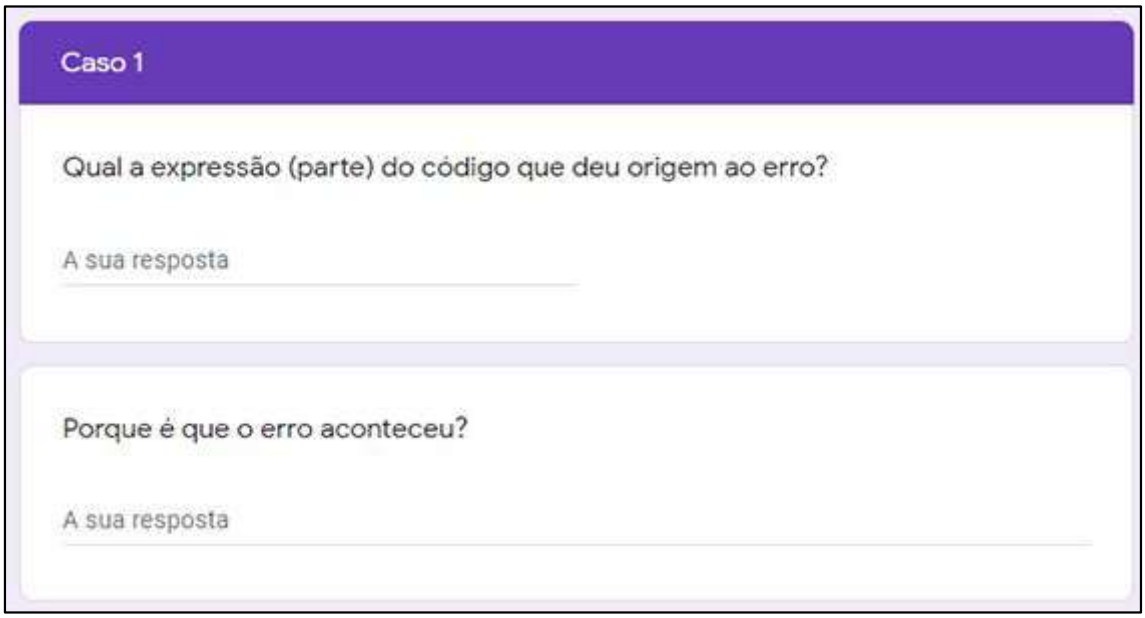

Figure B.1 - First two questions of method one

The open questions are the same for every method, so Figure B.1 covers these questions for every method and will not be repeated.

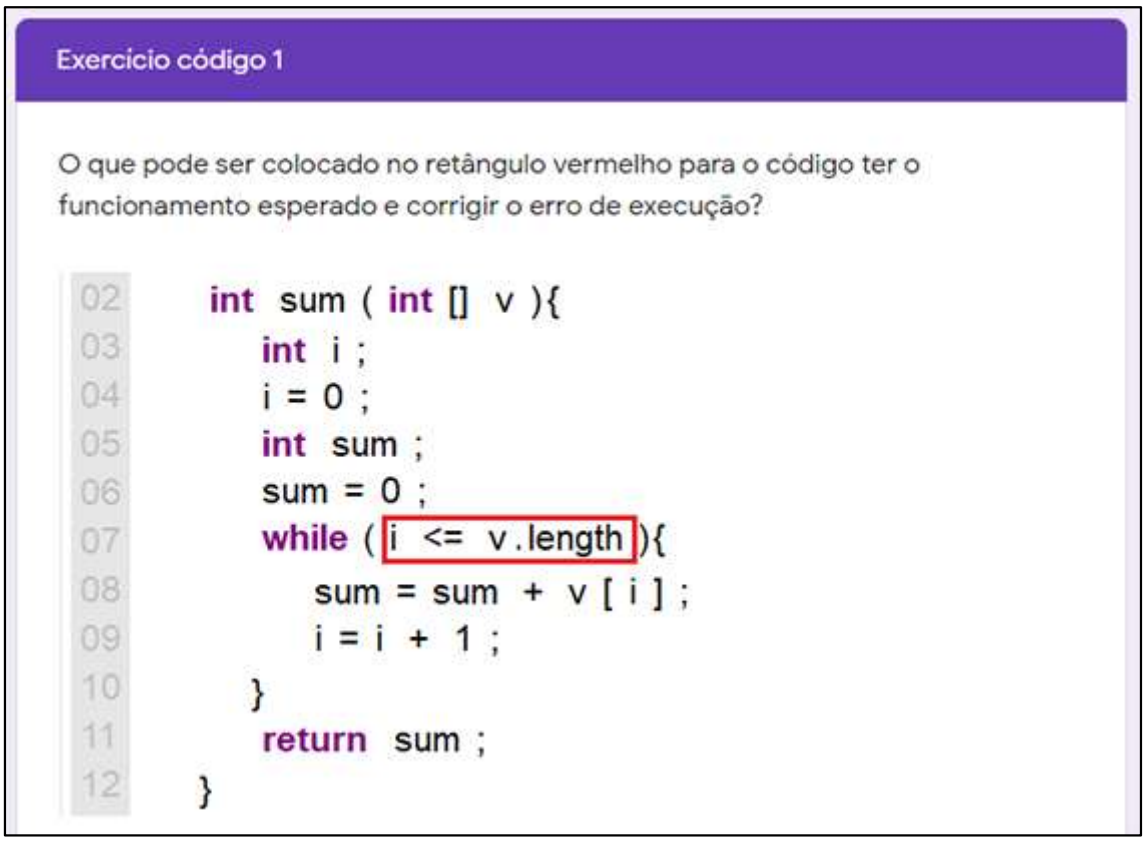

Figure B.2 - Multiple-choice question for the first method

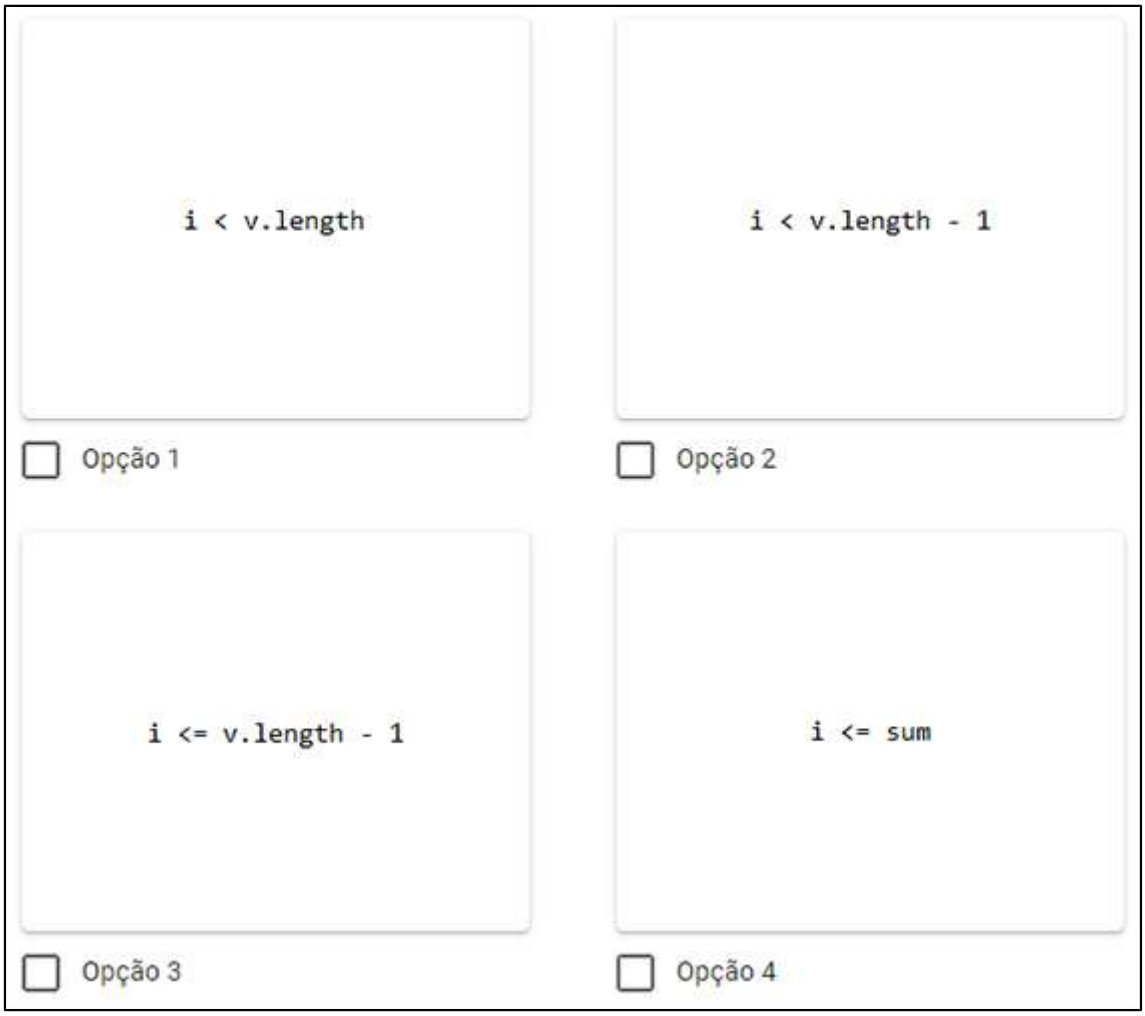

Figure B.3 - Options for the multiple-choice question of the first method (Answers: 1, 3)

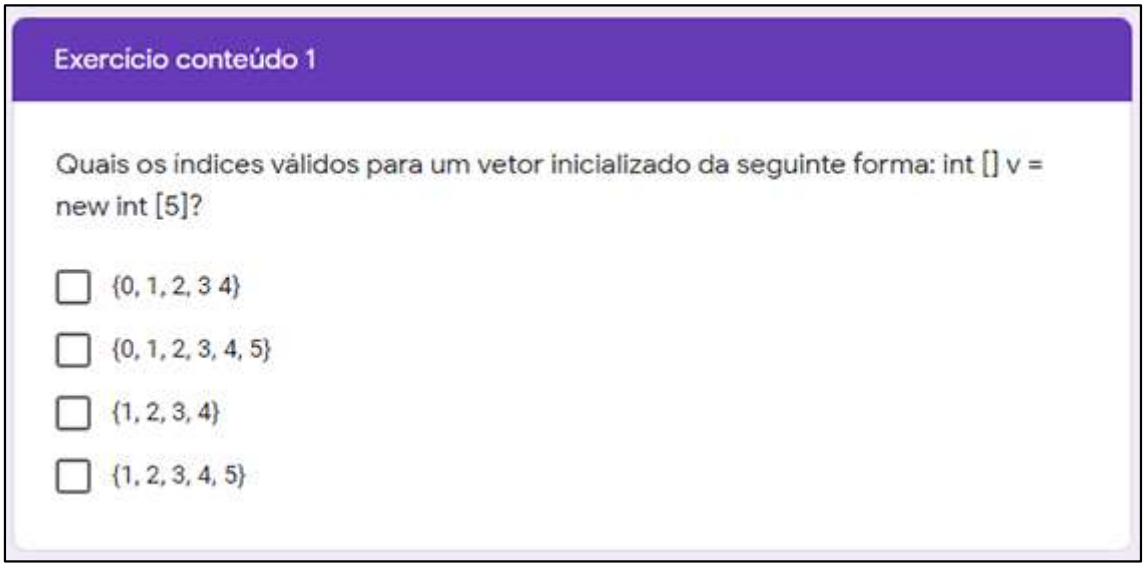

Figure B.4 – Last question of the first method (Answers: 1)

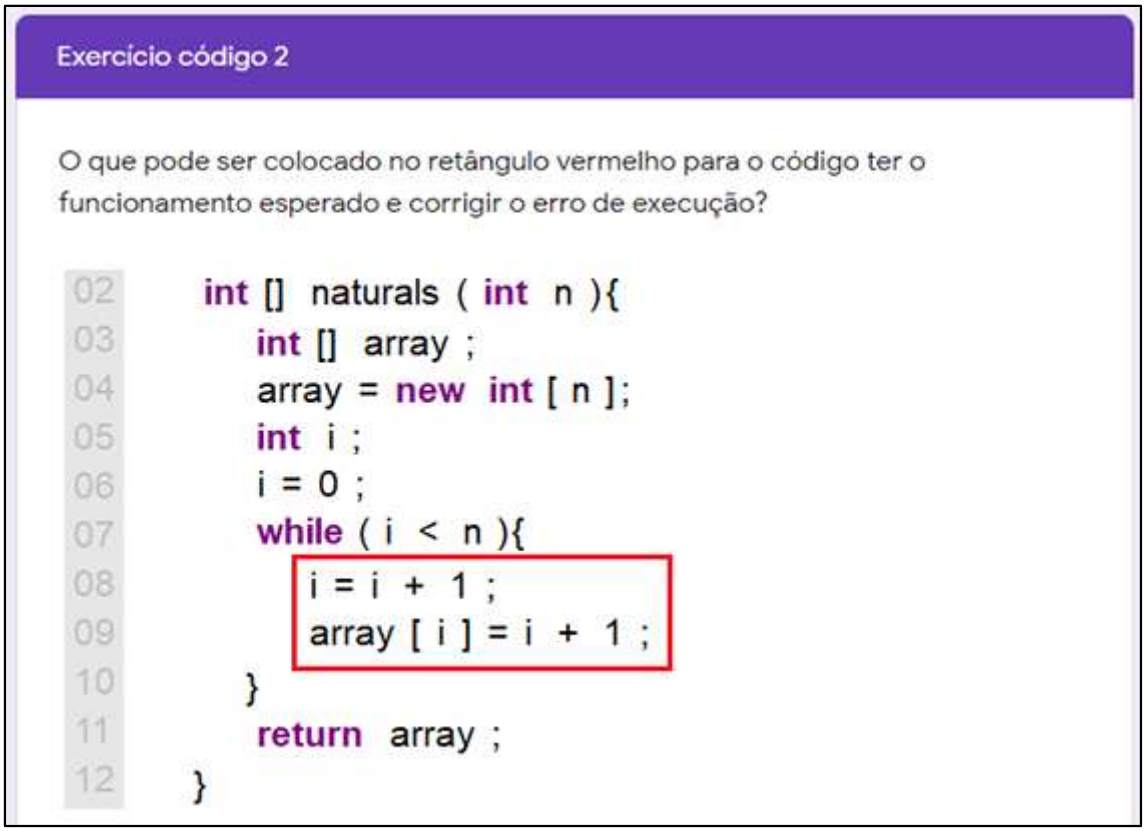

Figure B.5 - Multiple-choice question for the second method

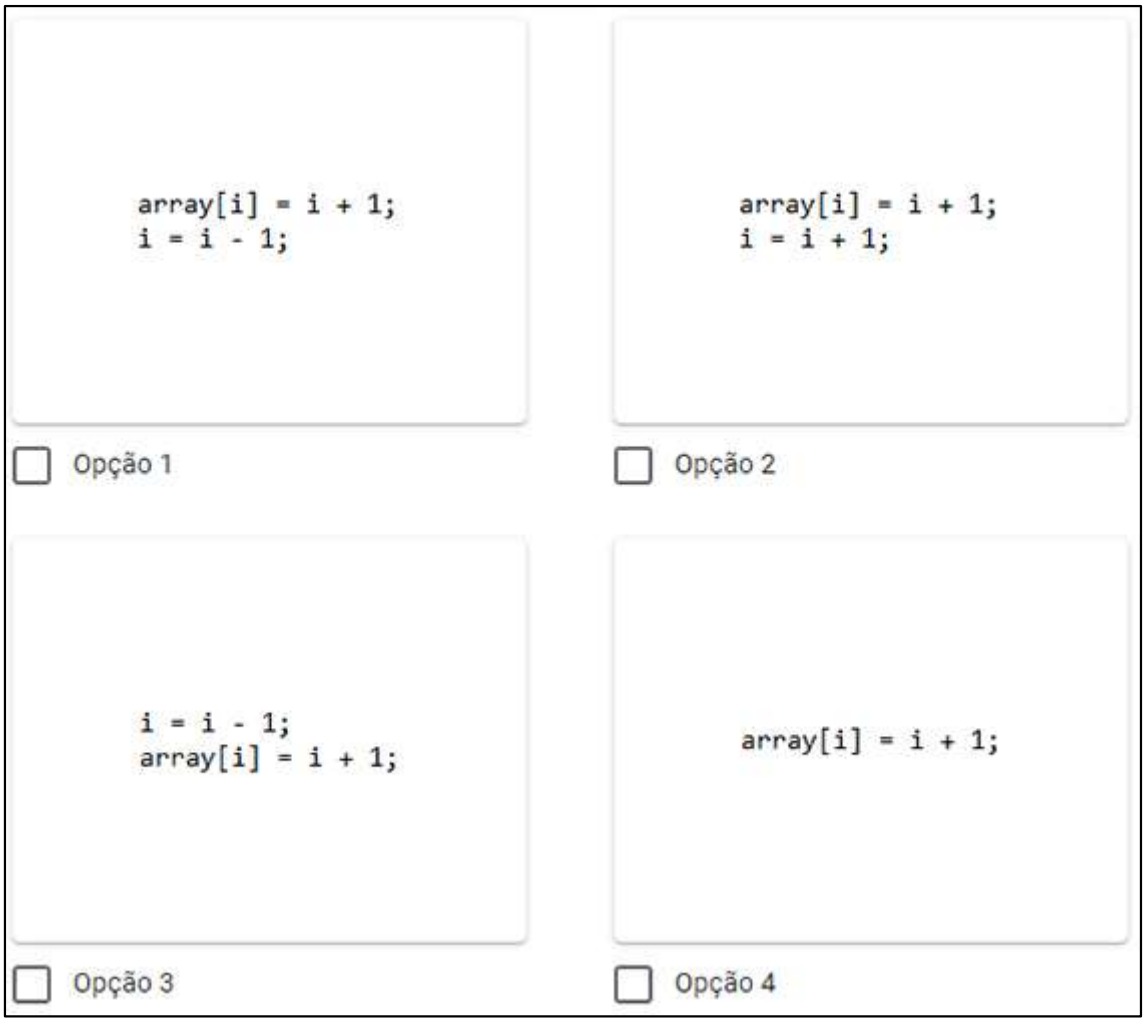

Figure B.6 - Options for the multiple-choice question of the second method (Answers: 2)

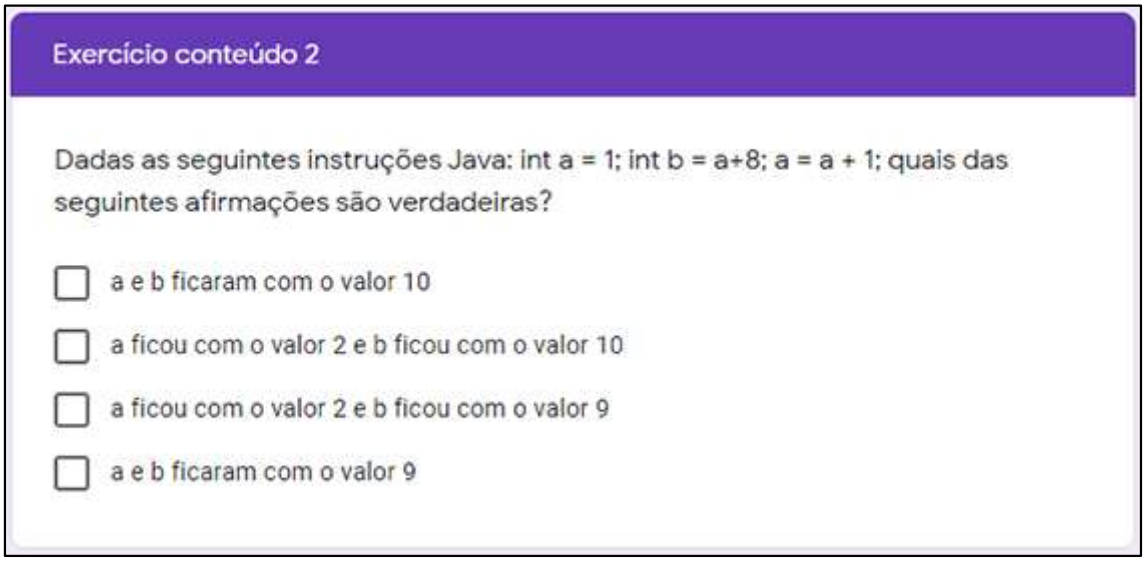

Figure B.7 - Last question of the second method (Answers: 3)

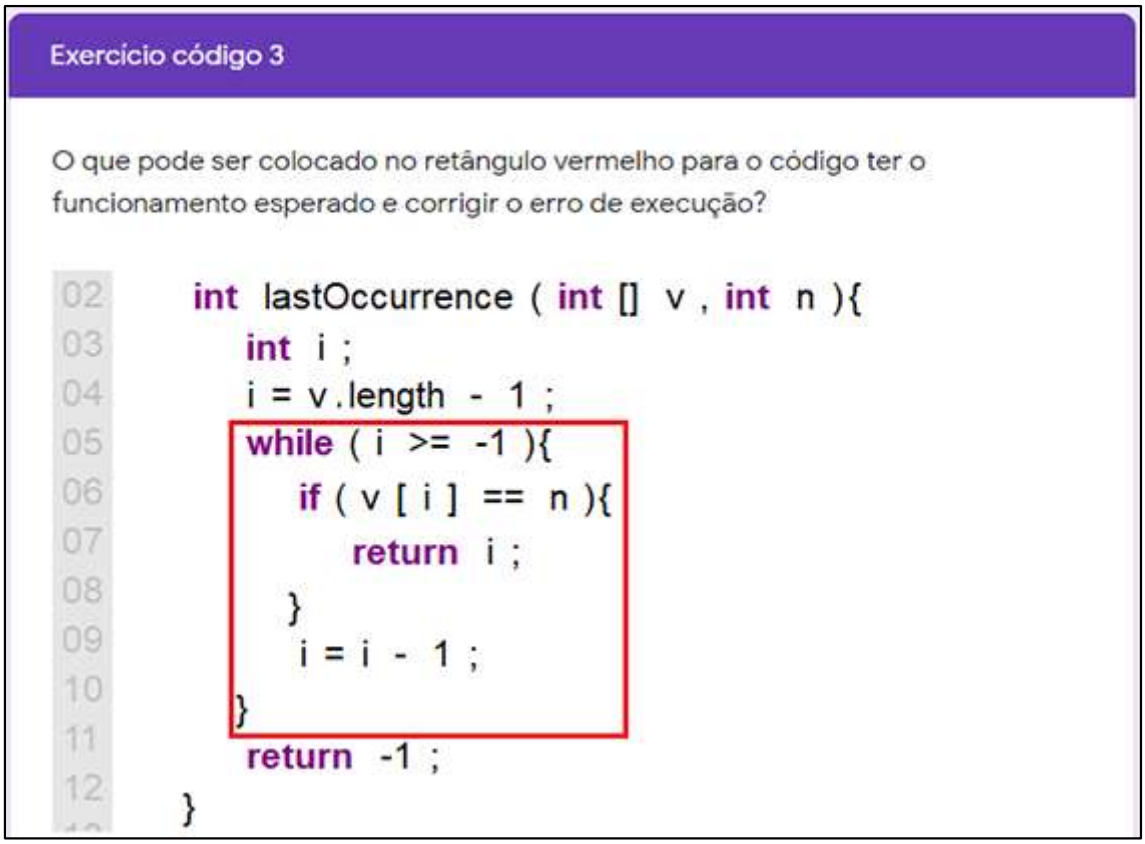

Figure B.8 - Multiple-choice question for the third method

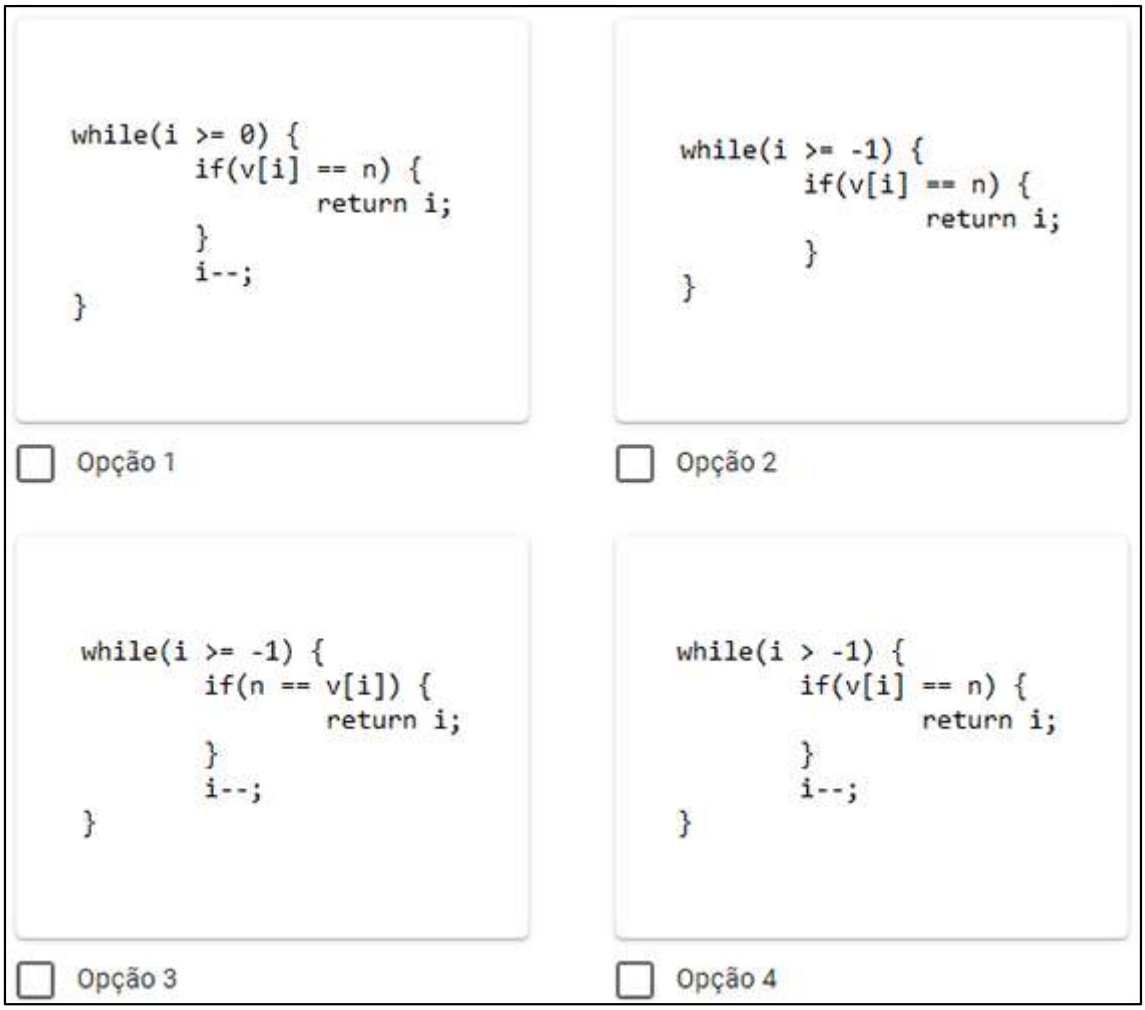

Figure B.9 - Options for the multiple-choice question of the third method (Answers: 1, 4)

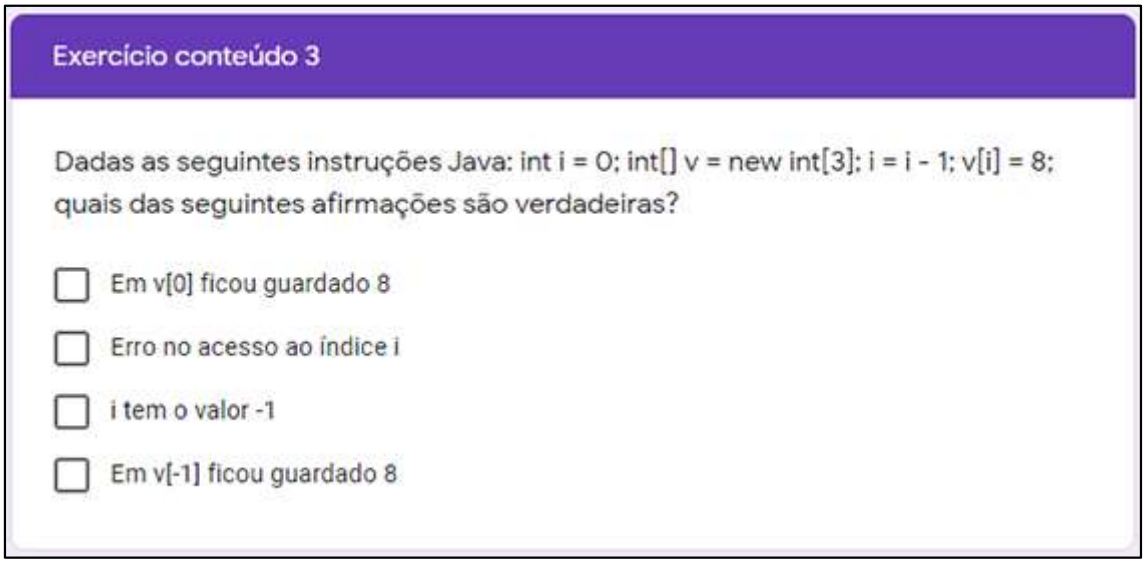

Figure B.10 - Last question of the third method (Answers: 2, 3)

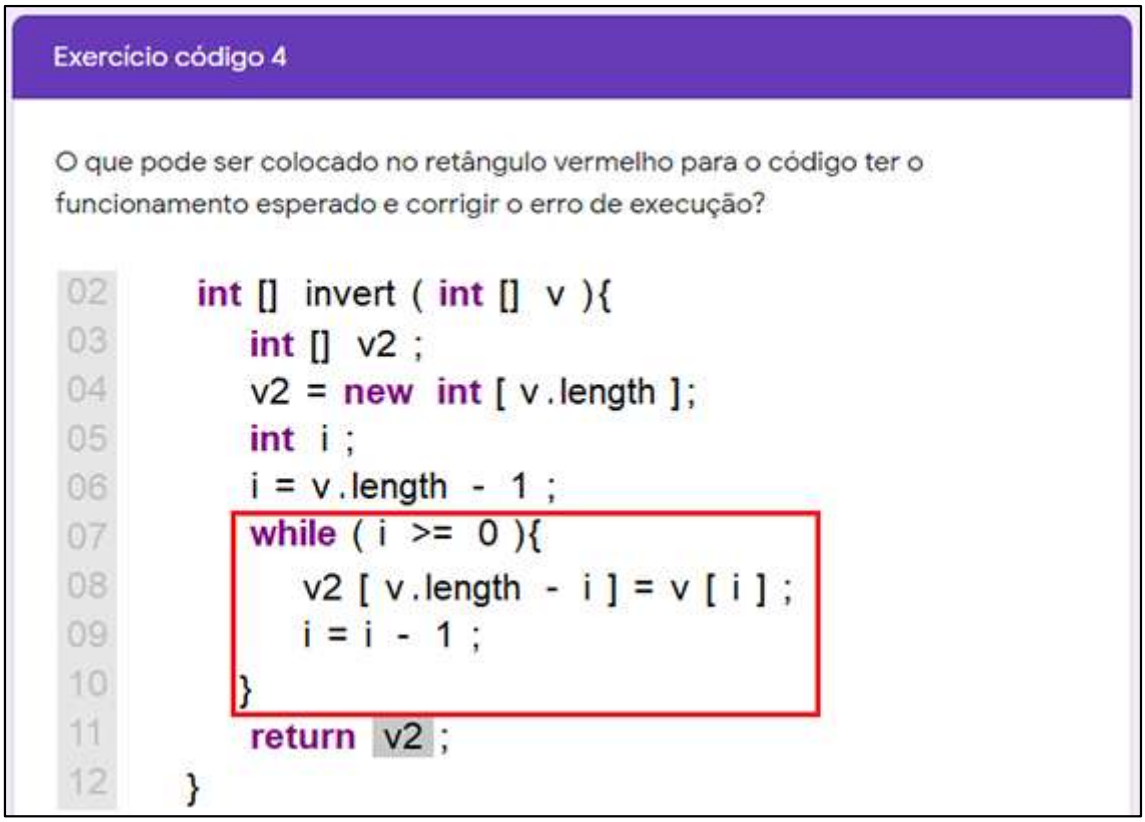

Figure B.11 - Multiple-choice question for the fourth method

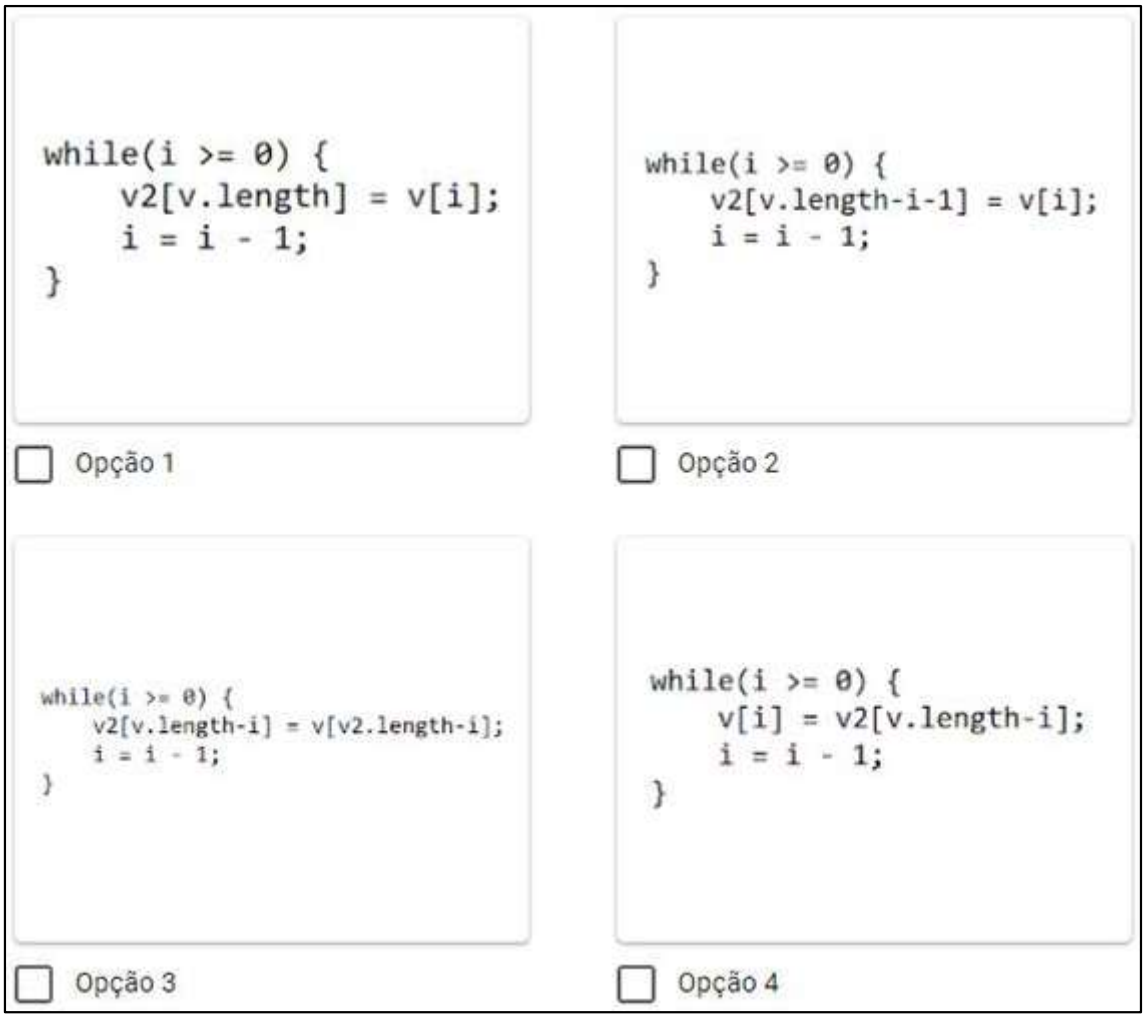

Figure B.12 - Options for the multiple-choice question of the fourth method (Answers: 2)

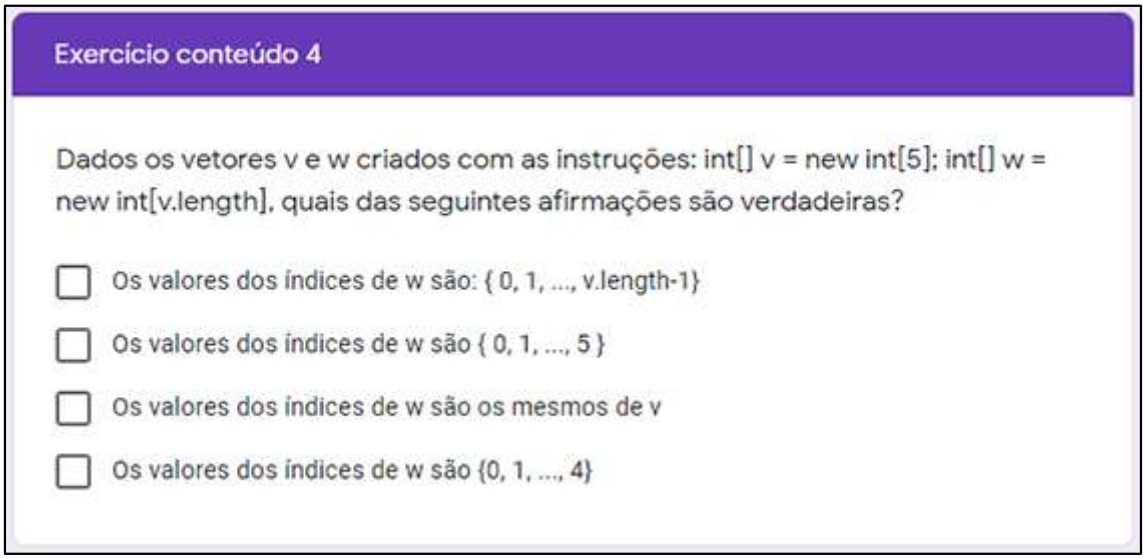

Figure B.13 - Last question of the fourth method (Answers: 1, 3, 4)

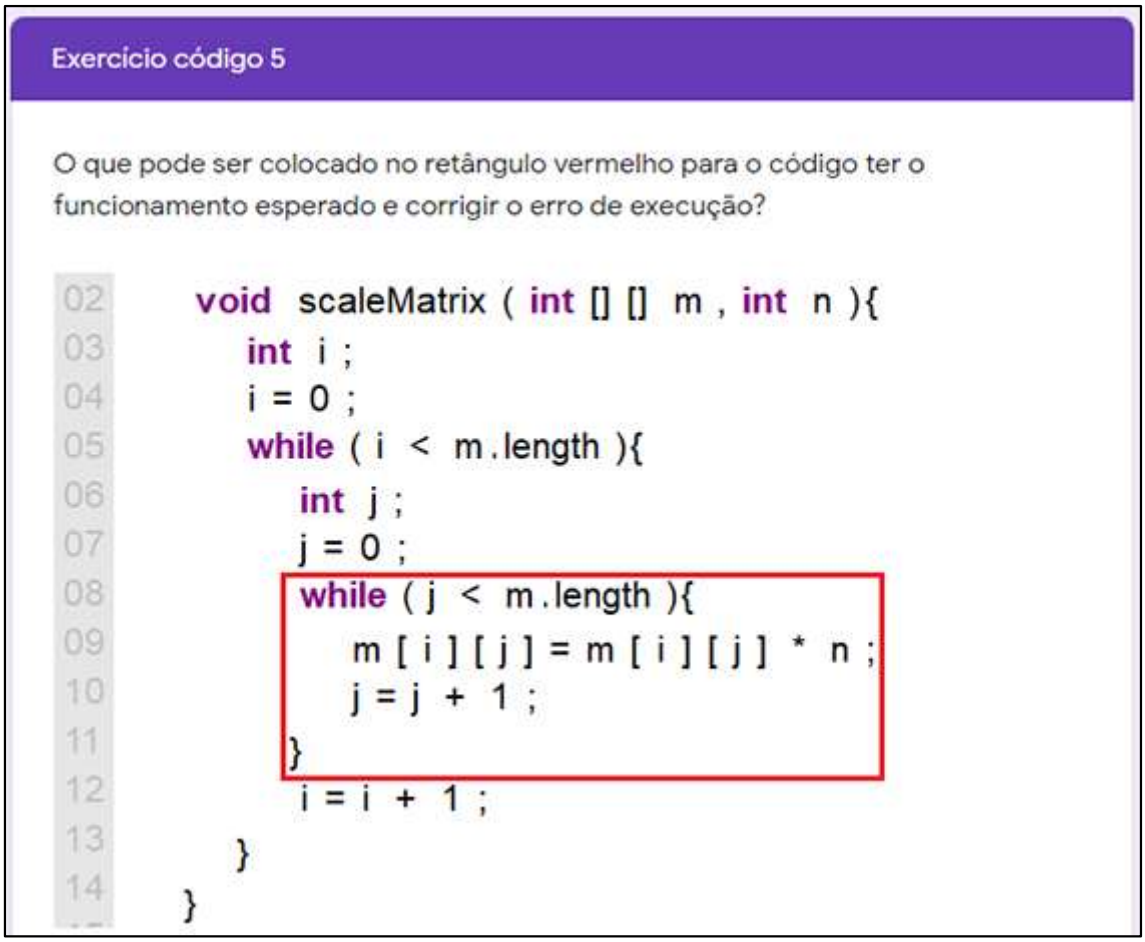

Figure B.14 - Multiple-choice question for the fifth method

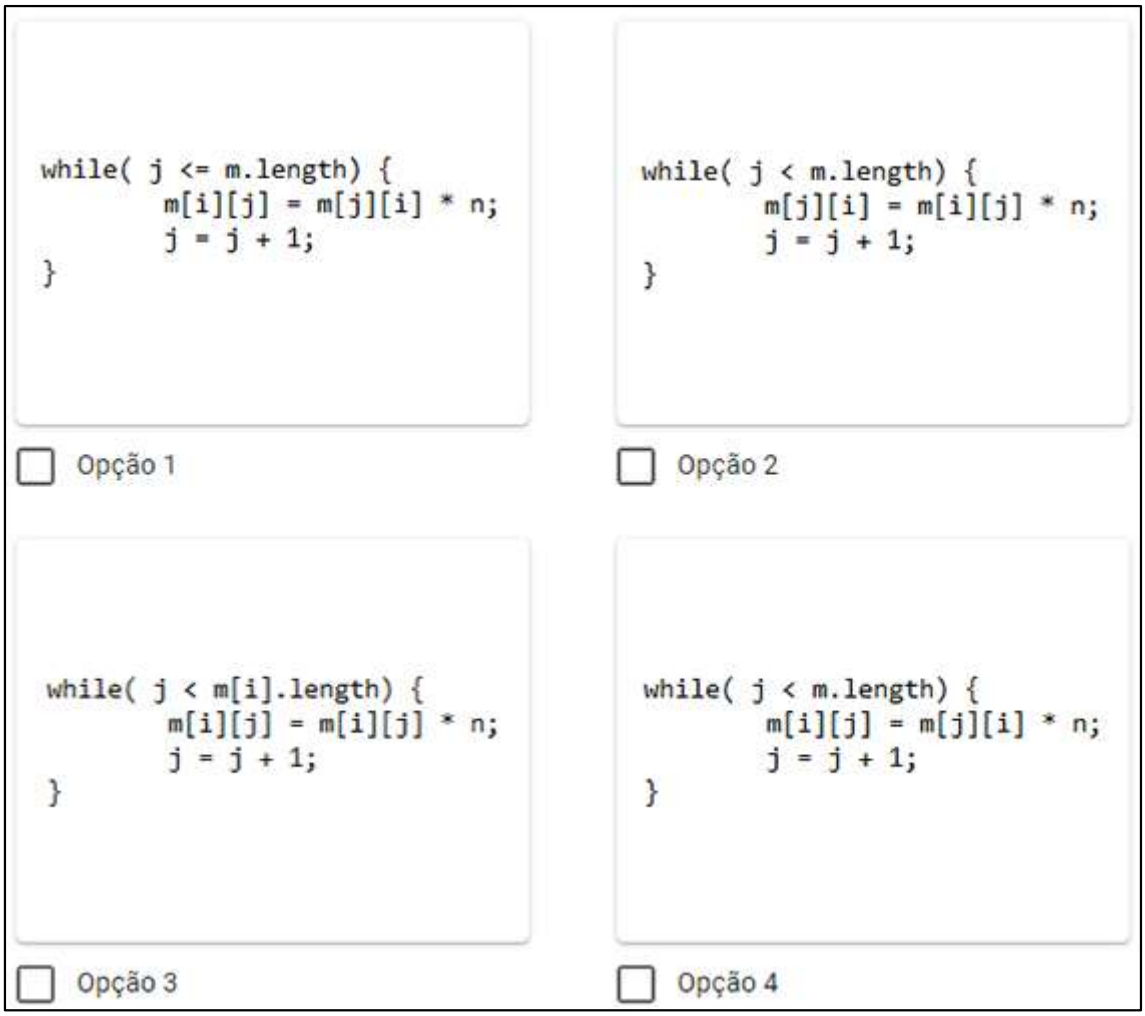

Figure B.15 - Options for the multiple-choice question of the fifth method (Answers: 3)

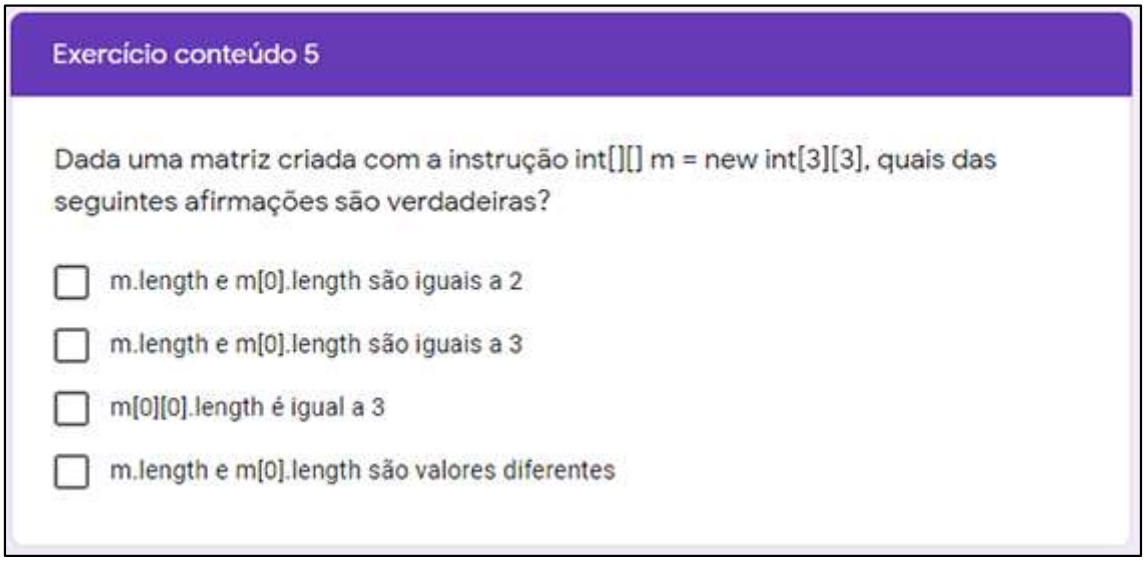

Figure B.16 - Last question of the fifth method (Answers: 2)

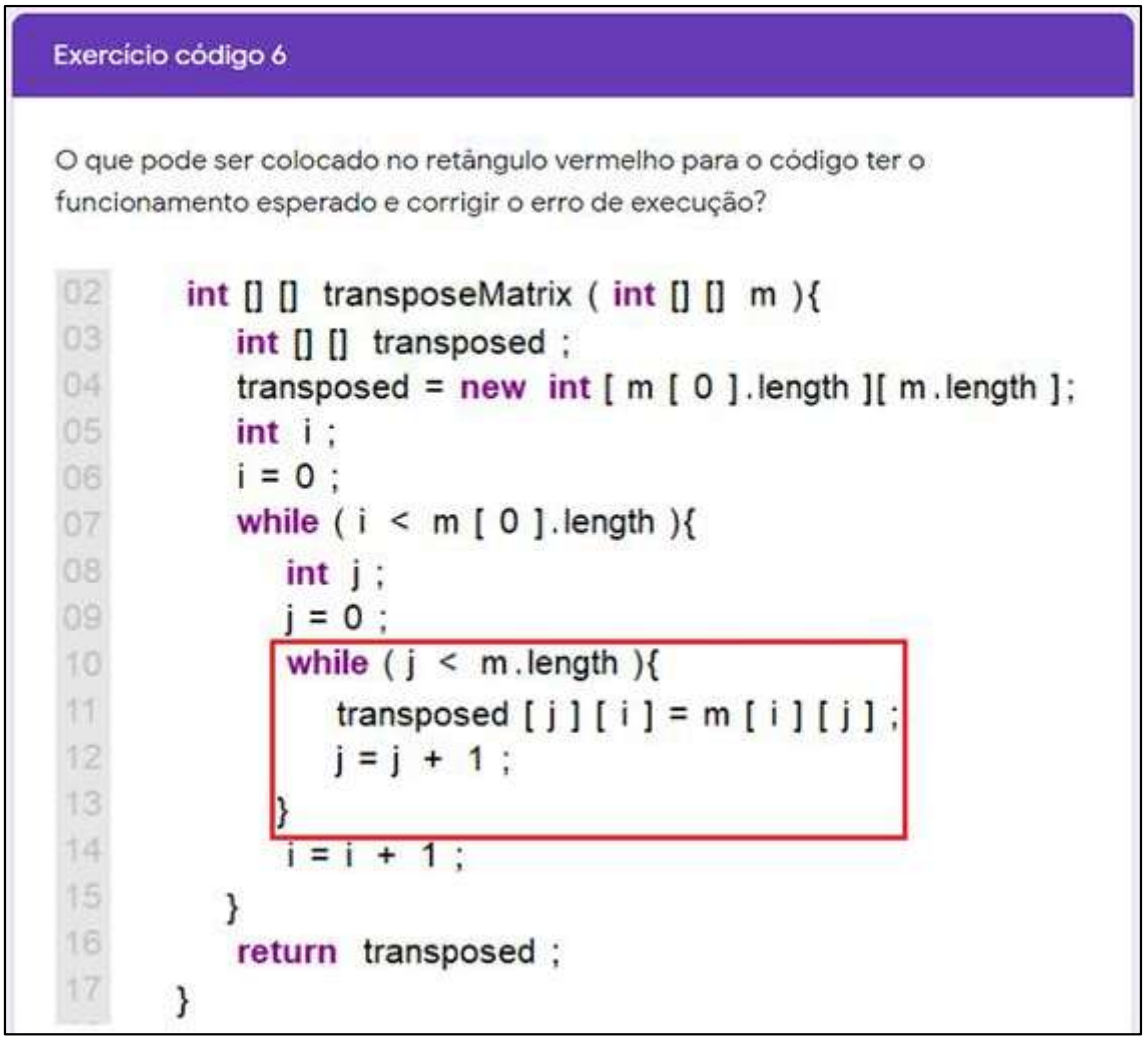

Figure B.17 - Multiple-choice question for the sixth method

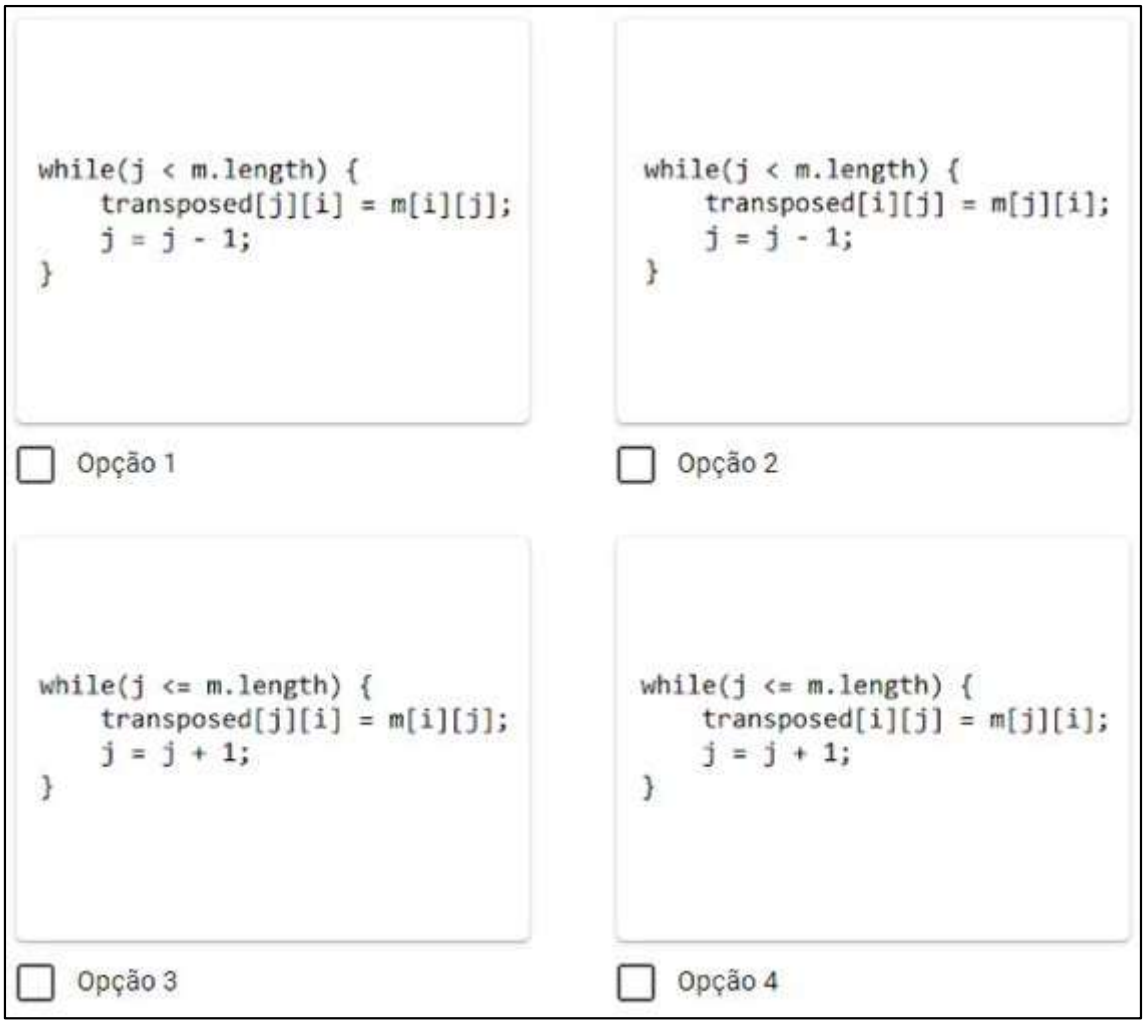

Figure B.18 - Options for the multiple-choice question of the sixth method (Answers: None)

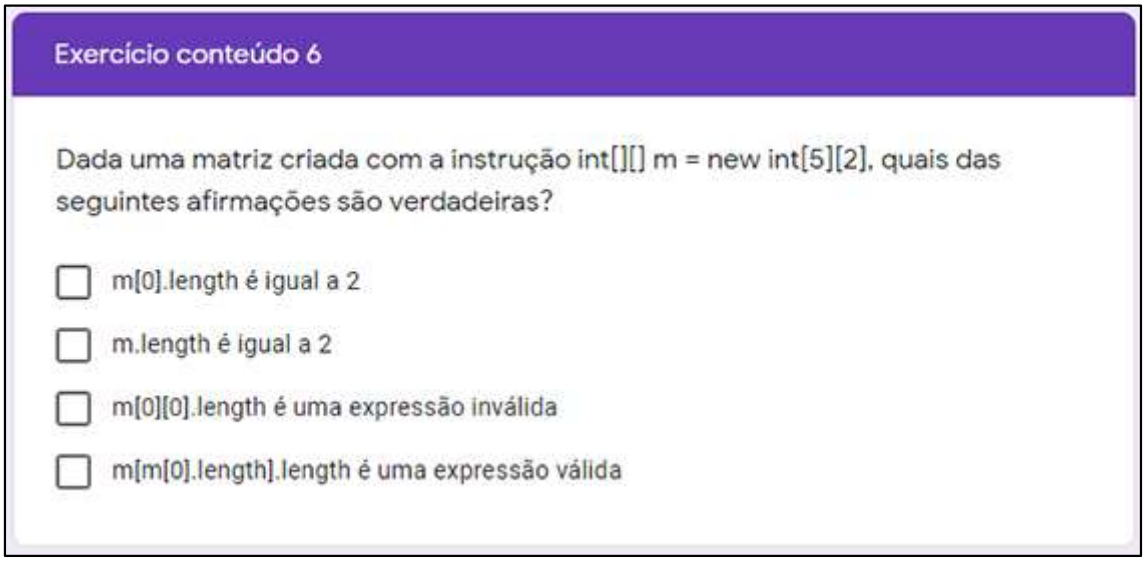

Figure B.19 - Last question of the sixth method (Answers: 1, 3, 4)