

Enhancing textual explanations for Java methods with variable role knowledge

Ricardo Cardoso da Silva

Master in Computer Engineering,

Supervisor:

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

Co-Supervisor:

Doctor Ricardo Daniel Santos Faro Marques Ribeiro, Associate Professor,
Iscte - Instituto Universitário de Lisboa

October, 2020

Enhancing textual explanations for Java methods with variable role knowledge

Ricardo Cardoso da Silva

Master in Computer Engineering,

Supervisor:

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

Co-Supervisor:

Doctor Ricardo Daniel Santos Faro Marques Ribeiro, Associate Professor,
Iscte - Instituto Universitário de Lisboa

October, 2020

Acknowledgments

First, i would like to thank my family as they have been very supportive during this last year. The experiences they shared with me proved to be most valuable in overcoming difficulties. A special thanks to both my parents, they watched over me throughout my whole life and taught me how to have a good head above my shoulders, and for that, I am deeply grateful.

I would like to thank all my friends, the time spent with them helped me relax and not get anxious whenever a new obstacle would appear.

I would like to thank each person who took the evaluation questionnaire and set aside a considerable amount of time to give detailed feedback.

Lastly, i would like to give a special thank you to both my supervisors, André Santos and Ricardo Ribeiro. Throughout this last year, they guided me to not stray onto a bad path, gave me advice on how to improve this project, and properly criticized my undoings even during the current outbreak. Without their insight, this project would probably never come to fruition.

Resumo

Durante as suas fases iniciais de aprendizagem de programação, os estudantes naturalmente vão deparar-se com vários obstáculos no seu percurso para compreender os vários conceitos de programação. Apesar do professor ser responsável por ajudar nestas situações, alunos ainda têm dificuldades para entender os conceitos básicos de programação e uma baixa compreensão destes pode arriscar futuros projetos dos alunos. Para melhorar esta experiência de aprendizagem, esta dissertação apresenta um protótipo que pode traduzir métodos básicos numa explicação textual, que também é enriquecido com o conhecimento de papéis de variáveis. Primeiro, com base numa experiência anterior, foram feitos dois estudos, o primeiro analisa o metadiscorso de como código é explicado e o segundo centra-se em entender como cada papel de variável influencia explicações de professores, estes servem de base para a estrutura da implementação. Para a avaliação, fizemos um questionário que contém explicações para diferentes métodos básicos e enviámo-lo a vários professores experientes para avaliar, de um a cinco em funcionalidade, completude e legibilidade. Os resultados mostram que metade das traduções recebem bons resultados, com maior parte dos votos sendo quatros ou cincos. No entanto, em métodos específicos, os resultados mostram que teve um baixo desempenho e sendo assim, o protótipo foi adaptado para ter um melhor desempenho nessas situações. Os resultados mostram que o conhecimento de papéis variáveis pode ser utilizado para melhorar componentes textuais e essas podem ser úteis para melhorar a experiência de aprendizagem inicial, pelo menos na perspectiva de um professor.

Palavras-Chave: Pedagogia; Programação; Interpretação.

Abstract

During their early stages of learning programming, students will naturally face various obstacles in their journey to understand the various programming concepts. Although the teacher is responsible for helping with these events, students still have difficulty understanding the basic concepts of programming and a low understanding of these can jeopardize students' future projects. To improve this learning experience, this dissertation presents a prototype that can translate basic methods into a textual explanation, which is also enriched with the knowledge of variable roles. First, based on a previous experience, two studies were made, the first analyzes the metadiscourse of how code is explained and the second focuses on how each variable role influences an explanation, these serve as the basis for the implementation structure. For the evaluation, we made a questionnaire that contains translations for different basic methods and sent it to several experienced teachers to evaluate, from one to five in functionality, completeness, and readability. The results show that half of the translations received good results, with most votes being four or five. However, in specific methods, the results show that it had low performance and so, the prototype was adapted to perform better in these situations. The results show that knowledge of variable roles can be used to improve textual components, and these can be useful to improve the initial learning experience, at least from a teacher's perspective.

Keywords: Pedagogy; Programming; Interpretation.

Index

Acknowledgments	i
Resumo	iii
Abstract	v
Index	vii
List of tables	ix
List of figures	xi
Acronyms and abbreviations	xiii
1. Introduction	1
1.1. Motivations and background	1
1.2. Research questions	1
1.3. Objective and approach	2
1.4. Document structure	2
2. Related Work	3
2.1. Difficulties for beginners	3
2.2. Variable Roles	4
2.3. Pedagogical Tools	5
2.4. Natural language	8
3. Analytical Studies	9
3.1. Metadiscourse study	9
3.1.1. Preparation	9
3.1.2. Tags analysis	11
3.1.3. Bigram study analysis	14
3.2. Variable roles study	17
3.2.1. Array Index Iterator	17
3.2.2. Gatherer	18
3.2.3. Most Wanted Holder	19
3.2.4. One Way Flag	20
3.2.5. Temporary	21
3.2.6. Most Recent Holder	21
3.2.7. Fixed Value	22
4. Prototype Implementation	23
4.1. Paddle	24

4.2.	Variable role identification	24
4.2.1.	Gatherer	26
4.2.2.	Fixed Value	26
4.2.3.	Stepper	26
4.2.4.	Array Index Iterator	27
4.2.5.	Most Wanted holder	27
4.2.6.	One Way Flag	28
4.3.	Additional Analysis	28
4.3.1.	Function Classifier	28
4.3.2.	Recursion	29
4.4.	Prototype Development	29
4.4.1.	Code Components	30
4.4.2.	FVParameterComponent	31
4.4.3.	Method Translation	32
4.4.3.1.	Bullet Points	32
4.4.3.2.	Expression Translator	33
4.4.3.3.	Method Translator	35
4.4.3.4.	Loop Translator	36
4.4.3.5.	Selection Translator	38
4.4.3.6.	Assignment Translator	38
4.4.3.7.	Return Translator	39
4.4.4.	Text Smoothing	41
5.	Prototype Evaluation	43
5.1.	Preparation	43
5.2.	Questionnaire Structure	43
5.3.	Results	45
5.4.	Prototype modifications	47
6.	Conclusions	51
6.1.	Main conclusions	51
6.2.	Future work	52
	Bibliography	53
	Appendix A	56
	Appendix B	65

List of tables

Table 2.1 - Descriptions of variable role.	4
Table 3.1 - Taxonomy adapted from [7] and each attributed color for the study.	11
Table 3.2 - Metadiscourse patterns identified from each explanation by each instructor.	12
Table 3.3 - Metadiscourse patterns instances per location.	12
Table 3.4 - Occurrences of each bigram sorted per location.	16
Table 3.5 - Array Index Iterator patterns and their occurrences.	18
Table 3.6 - Gatherer patterns and their occurrences.	19
Table 3.7 - Most Wanted Holder patterns and their occurrences.	19
Table 3.8 - One Way Flag patterns and their occurrences.	20
Table 3.9 - Temporary patterns and their occurrences.	21
Table 3.10 - Most Recent Holder patterns and their occurrences.	22
Table 4.1 - Relevant data structures of Paddle.	25
Table 4.2 - Roles related to each component.	31
Table 4.3 - Example of a translation for each element.	33
Table 4.4 - An example of each implemented variation for the first statement.	37
Table 4.5 - Examples of short explanations about each role.	39
Table 4.6 - Examples of assignment translations for each variation.	40
Table 4.7 - Examples of return translations for each variation.	41
Table 5.1 - Each method provided in the questionnaire and the concept of each portrays.	44
Table 5.2 - Results gathered from the first questionnaire.	46
Table 5.3 - All changes made according to the results of the evaluation.	49

List of figures

Figure 2.1 - BlueJ's main view.	5
Figure 2.2 - Students main view in ProPat.	6
Figure 2.3 - Illustration of a method <i>max</i> in PandionJ.	7
Figure 4.1 - Source code of the method being translated (left side of the view).	23
Figure 4.2 - Generated translation for the <i>summation</i> method (right side of the view).	23
Figure 4.3 - Example of Javardise's link function.	25
Figure 4.4 - Diagram about each step to generate an explanation.	30
Figure 4.5 - Example of bullet points before each line.	32
Figure 5.1 - Example and explanation showcase in questionnaire.	44

Acronyms and abbreviations

AI – Array Index Iterator

API – Application Programming Interface

CS1 – Computer Science 1 (first university-level course where programming is taught).

IDE – Integrated Development Environment

MWH – Most Wanted Holder

OWF – One Way Flag

SWT – Standard Widget Toolkit

SWUM – Software Word Usage Model

TED – Technology Entertainment and Design

UML – Unified Modeling Language

Introduction

1.1. Motivations and background

The early stages of learning programming are essential for students, given that in this timeframe students build their own comprehension about the inner workings of programming primitives and algorithms. Without proper guidance and attention, students may develop misconceptions [1] that not only add difficulties to their learning process but also may affect their future work negatively [2].

There are several tools to aid programmers in their struggle, but many of these tools are not developed with Computer Science 1 (CS1) courses in mind, since they often contain information that confuse and may misdirect beginners to undesirable paths. There have been pedagogical tools whose objective is to improve this learning experience, such as pedagogical debuggers, which try to improve their step-by-step analysis by adding visuals [3], pedagogic IDEs that focus on essential information and provide only the necessary elements for the learning process [4], or pedagogic plugins which use different features to make programming more intuitive [5], [6].

Nowadays, several CS1 courses have different teaching methods, and often teachers have to deal with large groups of students (it is not uncommon to have course editions with more than 100 students). This makes it hard for the instructor to pay attention to every issue, resulting in poor guidance and consequently develops misconceptions in the knowledge of students [6]. To properly teach many students at the same time, regularly, examples of code are used to convey every essential aspect of the initial programming experience. By teaching one example at a time, teachers can restrict and minimize possible misconceptions.

PandionJ [7] is a pedagogical debugger that builds on the concept of variable roles to provide explanations based on visual portrayals of the algorithms to achieve the objective of increasing student's misunderstanding of program dynamics. Continuing to explore this approach, we believe that it is also worth to pursue and research how impactful would be if explanations are delivered using natural language while enhancing them with the concept of variable roles.

1.2. Research questions

1. How are variables with different roles explained by instructors using natural language?
2. How helpful is the knowledge of variable roles on a static explanation using natural language?

3. Is it possible to automatically generate meaningful explanations of small functions in natural language?

1.3. Objective and approach

Our main objective is to develop a prototype of a pedagogic tool capable of analyzing examples of code used to teach essentials in CS1 courses and generate an explanation in natural language describing introductory Java methods, in a way that offers automatic guidance to novices to put them on the right path or relieve them of misconceptions. Additionally, we want to explore whether or not variable role knowledge can be helpful in strengthening the textual explanations, therefore we are going to identify each variable's role and see how it can enhance each explanation.

To provide a static analysis of a method using variable roles, first, it is necessary to acquire detailed information about method bodies, which requires static analysis. In order to do this, we are going to use a recent framework by the name of Paddle that is under development.

Another objective of this dissertation is to discern how these variable roles are explained using natural language. In order to achieve this, a previous pedagogical study, where programming instructors are asked to explain Java methods through words and illustrations, were analyzed to determine the most notable patterns when explaining each variable role. These served as a basis to design the content of the automatic explanations.

1.4. Document structure

This chapter supplies an overview of what is going to be discussed across this dissertation. The following chapters will provide:

- Chapter 2 focuses on reporting relevant previous research related to this dissertation.
- Chapter 3 presents an analysis of previous research, focusing on discursive patterns of both metadiscourse and variable roles
- Chapter 4 encapsulates all choices of implementation, from how each role was identified to how the translation is conceived.
- Chapter 5 presents our evaluation method, discusses results, and what modifications were made according to those.
- Chapter 6 discusses our conclusions about the research questions and future developments.

Related Work

2.1. Difficulties for beginners

In 2005, a study was done with people with programming experience [8]: 559 students and 34 teachers. They had to answer a questionnaire based on background, course contents, and aspects of learning. The objective of this study was to evaluate a pattern of difficulties among the testers. Results show some difficulties when dealing with recursion, pointers, or references, mostly by the students. Lahtinen et al. stated that “novice programmers do not seem to be the understanding of basic concepts but rather learning to apply them”, which may indicate an underlying problem with their respective CS1 courses program or learning methods.

Qian et al. [1] using many perspectives from several references, compiled a report about student difficulties. According to the authors, novices harbor misconceptions which may affect their future developments. Mistakes that novices make vary from simple syntax mistakes to lack of strategy when planning the structure of a program. To properly review the misconceptions, every mistake was categorized under the lack of one of three types of knowledge, these being: syntactic knowledge, conceptual knowledge, and strategic knowledge.

Syntactic Knowledge refers to the grammatical rules of the programming language. Normal syntax mistakes [9] refer to missing semi-colons or using variables without declaring them, resulting in syntax errors. Furthermore, studies also show that some syntax errors require more time to solve than others [10].

Conceptual Knowledge is the understanding of an individual about the code’s inner working: what happens in each assignment, where each value is stored, the consequences of conditionals, etc. Although syntax errors are far more common, conceptual errors take more time to fix, as they might be the consequence of misconceptions held by the individual. An experiment by Simon [11] showed that students still struggle with assignments or a sequence of them even after some programming courses.

Strategic Knowledge refers to the knowledge required to plan or strategize how to develop efficiently. Several actions in programming require this knowledge. From developing a single method to a full program, debugging productively, or predicting and avoiding bugs. Qian et al. [1], [4] said that “only knowing syntax and semantics does not necessarily produce a good programmer”. The lack of proper strategy and planning to program may lead to many problems, among some being, bugs, rewriting, or more high-level problems like deadlocks.

2.2. Variable Roles

Variable roles refer to an empirical study by Sajaniemi [12], by analyzing the roles of variables of 109 novice-level programs developed by experts they were able to introduce a list of roles which encapsulates 99% of the roles of variables in novice programs. Table 2.1 displays a list and description of these roles.

Table 2.1 - Descriptions of variables roles.

Roles	Description
Fixed Value	After being assigned a value, that value will never change
Follower	Is assigned a previous value from another variable
Gatherer	Is an accumulation of several individual values
Most Wanted Holder	Holds the value that most respects a certain condition
Most Recent Holder	Holds the latest encountered value while going through a succession of values.
One Way Flag	Is assigned a value and it does not change again after the second assignment
Organizer	An array storing elements to be reorganized
Stepper	Goes through a predictable succession of values
Temporary	Is assigned a value for a short time only

Under a pedagogical view, a study was made about a review where three different courses undergo changes to improve their program [13]. Where previously they used Java, C, and Delphi as an introductory language. They started to use Python while focusing on variable roles. Although there was some struggle dealing with roles like Most Wanted Holder or Stepper, results for all three courses were mostly positive. Even though the positive results might be from the language change. Teachers have stated that they could more easily convey concepts by focusing on variables roles. Not only for the teacher but the students as well, by holding the same concepts of variables roles, they better understand the viewpoint of the instructor figure. Furthermore, an increase in motivation and desire for learning was acknowledged, due to the success of students in completing assignments.

Another study [14], in an introductory Pascal course, separated students into groups and used different the learning process of variable roles for each group to compare the results. The groups were the traditional group, following the traditional course, the roles group, where the roles are taught, and the animation group, which used roles in conjunction with the animator. Results show that knowledge about variable roles allow for students to process information in a way of what is thought of as good code comprehension.

To further research this concept, experimental tools started to be developed around variable roles, such as the case of PlanAni [15], an animator used in a previously mentioned study [13]. And

more recently, a program by the name of PandionJ [7] was developed, a pedagogical debugger which implements variable roles concept in a step-by-step analysis (further explained in Section 2.3).

2.3. Pedagogical Tools

Over the years, the difficulties and misconceptions of beginners in programming through various studies were identified and acknowledged. To further research solutions to aid in introductory struggles, several pedagogical tools were developed.

BlueJ is a pedagogical IDE with development centered around object-orientation [4]. According to the authors, teachers find that teaching object-oriented is more difficult than procedural programming. After analyzing limitations in other environments, it was noticed that most environments are not object-orientation friendly or were too complex for beginner level. BlueJ was developed to offer an object-orientation friendly design. Its main view uses Unified Modelling Language (UML) to display relations between objects to better understand all interactions allowing users to constantly focus on the structure of the program (as shown by Figure 2.1). Tasks like creating an object are greatly simplified by making it accessible by a right-clicking a menu and displaying a window to input basic object information such as parameters or name. By maintaining a simple and objective visualization not only it allows for teachers to have an easier time explaining concepts, but also for beginners to quickly adapt to the new environment. However, the authors also theorize a dependency setback on their system. Students may grow overly dependent on the beginner-friendly environment and struggle when changing to other IDEs.

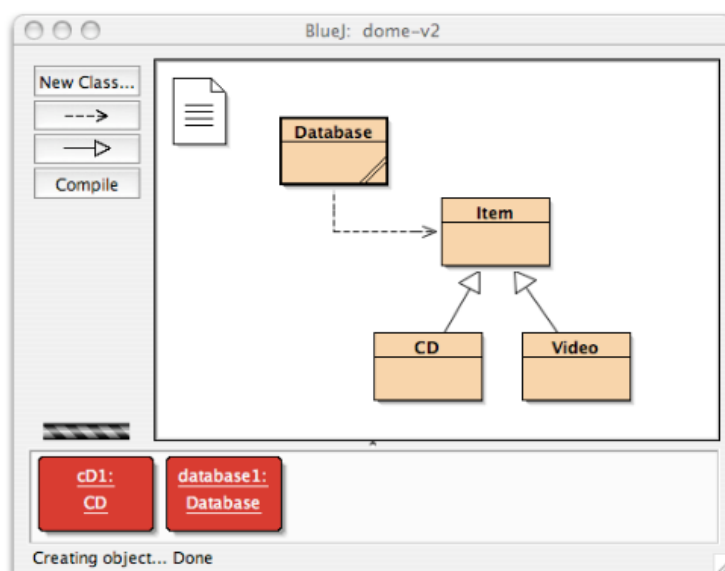


Figure 2.1 – BlueJ's main view.

Results with BlueJ [4] display that it was already adopted as introductory IDE by more than 800 universities. Opinions of students manifest that BlueJ is easier to use than other IDEs and its environment was helpful. Furthermore, reports also show that feedback from teachers is mostly positive. The conclusion by the authors is that although there are positives aspects in this project, more research on new approaches still needs to happen to furtherly fine-tune the subject of learning object-orientation programming.

Previous research shows that experienced programmers sometimes apply old solutions from their past experiences to conceived solutions to their current problems. However, beginners do not have the experience or knowledge to enact this type of problem-solving. ProPat [16] is an IDE for Eclipse inspired by this idea and tries to teach usual problem-solving patterns in introductory courses. It offers two different perspectives. The Teacher perspective, to allow the teachers to create exercises for the students to solve and add new possible patterns to solve the problem. Furthermore, with additional information inserted by the teachers, it allows to pinpoint some mistakes and provide an explanation. The Student perspective displays many views that allows the students to select an exercise to solve with no help or use one of the provided patterns to achieve the same objective. In case of difficulty, one of the views displays a description, specifications, and already made tests for an exercise. Figure 2.2 shows a detailed view of the Student perspective. According to the authors, this system allows for a student to intuitively associate a previous problem-solving pattern to a current problem.

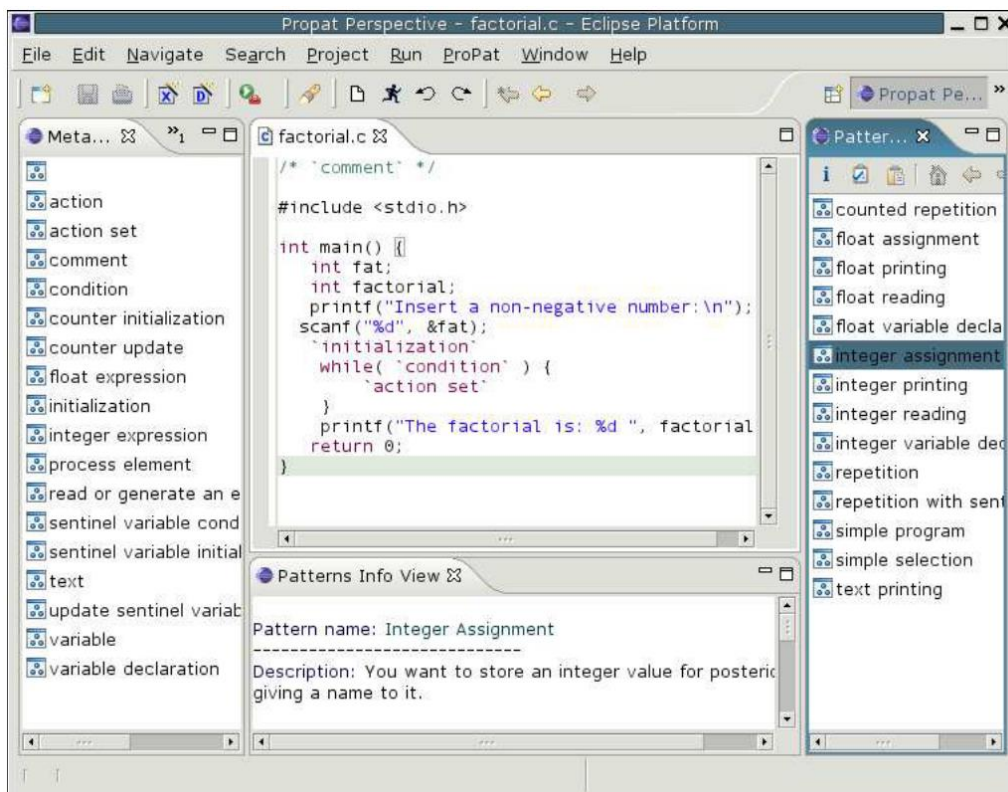


Figure 2.2 - Students main view in ProPat.

PandionJ is a pedagogical debugger developed in recent years as an Eclipse extension [7]. It focus on providing visual illustrations to students while debugging to facilitate the comprehension of code. Furthermore, it also uses variable role knowledge to enhance the visual experience.

To be able of identifying variable roles, a first static analysis of the source code is executed. A set of rules was created for each role. A variable would need to have all pre-defined conditions true to be classified as a role. To determine these conditions, there were functions developed to get essential method information, such as verifying if the variable is a parameter or if an assignment is inside a repetition control structure. In our work, although the approach is similar, some changes were made. A static analysis and pre-set of rules will still be done, but in this case, it is using the in-developing framework, Paddle, to access information of source code more easily.

To achieve another of their objectives, they realized an exploratory study [17] with six programming instructors. In which each of the instructors was filmed and asked to explain basic methods covering essentials in introductory programming by freely writing and speaking and after that to provide feedback. Using this study, they noted each pattern to depict each role and took note of the most prominent. Results show some similarities in illustrations by different instructors. This study is going to be the basis of our research, although in our case, it is more focused on the natural language patterns instead of illustrations.

PandionJ follows a usual step-by-step debugging analysis but offers illustrations. Each illustration varies depending by the variable role. Making it more intuitive for students who face struggles in comprehending the source of the error. Figure 2.3 is an example of a depiction of a method to find the highest value in an array, max being a Most Wanted Holder.

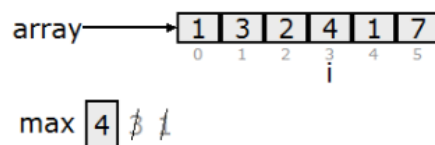


Figure 2.3 - Illustration of a method max in PandionJ.

2.4. Natural language

Nowadays, natural language keeps on being one of the most researched topics. As it is essentially our main form of communication, it is still investigated on several ways to use this knowledge to facilitate programming tasks.

One of the most common examples of natural language present in programming are comments, when dealing with large scale projects these are essential to ensure everyone is on the same page. As so, several ways of automatically generating comments were investigated such as Sridhara et al., who proposed an approach of identifying the most relevant segments of a Java method and focus the generation on those main points [18] which later was improved to do the same with parameters [19], high-level actions in methods [20] and classes [21]. Another approach was proposed by Wong et al. which consisted in analyzing already existing software repositories and generate comments based on those results [22]. It was also researched on how the use of neural networks could achieve this same objective as showed by Liang [23] and Hu et al. [24]. Similar to this project, Farinha attempted to generate descriptions for Prolog programs [25] yielding some good results. Another project, specifically for identifiers, was conceived to convert identifiers composed of 2 or more words into their complete textual version without abbreviations [26].

However, natural language is not always the end result, recent research shows how pure text can be the basis of beneficial generations to users. Deeptimahanti was able to, through NLP, build a system that converts natural language requirements into UML diagrams making it easier to analyze them [27]. Mandal developed a model that can convert textual mathematical problems to its source code version and discover the answer for said problem, although it had some flaws, they were able to achieve a high percentage accuracy ratio in generating right answers [28].

Analytical Studies

Our objective is to be able to create an explanation for different types of introductory programming functions/procedures. An important note is to understand which kind of text structure should be used, what linguistic patterns are most used in each explanation.

This chapter contains two analytical studies. The first study focuses on analyzing metadiscourse patterns to discover which are the most prominent. The second study, which is directly related to our first and second research questions:

- How are variables with different roles explained by instructors using natural language?
- How helpful is the knowledge of variable roles on a static explanation using natural language?

The latter analyzed a bundle of diverse explanations, to investigate if there are linguistic patterns directly connected to each variable role to further enhance our explanation.

3.1. Metadiscourse study

3.1.1. Preparation

To properly generate an explanation that can guide students through their obstacles, first we need to figure out what would be an acceptable text structure. Different students are compatible with different ways of explaining, whether they react better through a practical exposition of the method or a more theoretical approach, it must be acknowledged that our objective is to try to appease the most of every type of student. To achieve this, we researched what kind of text structure would be the most helpful through an analysis of a previous study in which 6 programming teachers were asked to explain a set of novice-level methods [17] and were recorded in the process. Each instructor was asked to verbally explain methods and in doing so, they complemented each explanation with drawings to better convey some concepts. The mentioned study's objective was to see how each instructor would depict concepts related to each variable role. Our approach will be more focused on the textual aspect, we will be analyzing discourse patterns and try to apply any findings to generate a properly structured explanation (all methods included in appendix B).

In order to unveil any patterns, we were inspired by a previous experiment [29]. To create metaTED, a corpus of metadiscourse for spoken language, a set of TED talks was randomly chosen, and by adapting a taxonomy from Ädel [30], they conducted a crowdsourcing study [31] to accurately annotate each segment of each TED talk. In the same fashion, we are going to apply the same method

to previously mentioned videos and annotate and analyze any frequently used metadiscursive patterns, and which of those will possibly enhance our future generated explanation.

The taxonomy used for the crowdsourcing experiment consisted of 16 categories listed in Table 3.1. These categories were all adjusted for the annotation of the TED talks and contrary to them, the videos to be analyzed are not structured in any way. The teachers were only asked to explain each method without any previous idea of what method and were not restricted in time or explanation styles. Hence, as the teachers used free speech, discourse among all videos is varied and might contain anomalies such as pauses, repetitions, or even mistakes. As such, there are some categories which we did not consider due to the difference in circumstances between studies. In that manner, arguing (ARG) and comment on linguistic form (COM) were not pondered during the annotation. ARG instances happen when the speaker tries to engage with the audience, in this case, there is no audience to be engaged with and COM, in this practical case, has a very low chance of happening, so both were discarded, leaving us with 14 categories.

To accurately annotate each category, the audio from the videos needed to be converted to raw text. So, to achieve this we used the Python Speech Recognition Package [32] and the Google Speech Recognition API to convert audio to Portuguese text. However, as already expected, the automatic conversion API was not enough to detect and convert all audio. There were instances in which the API would completely fail to detect some words correctly and be unable to convert sections. For this reason, in segments in which the API would fail, they would be properly mended by hand, while being as faithful as possible to the original audio.

Our chosen annotation approach was rather simple. For each identified occurrence of a category, either the color of the background of the sentence or the letter color was changed according to each category. Table 3.1 shows which color is correlated to each category. For categories with no occurrences, there is not an attributed color.

In the previously mentioned study, four of six instructors explained 7 different methods, with the remaining two having explained 6 methods. After the conversion of audio to text, we gathered a total of 40 different transcriptions. The transcriptions numbered a total of 16171 words, with the average per transcription being 404,3 words. Among all explanations, 1117 words was the highest number used in a single explanation and 142 was the minimum. Additionally, the highest difference of words between explanations for the same method was 868 words.

Table 3.1 - Taxonomy adapted from [7] and each attributed color for the study.

Tags	Description	Annotation Color
ADD	Adding to the topic with Asides	N/A
ANT	Anticipating response	N/A
ARG	Arguing	N/A
CLAR	Clarifying	Gray
COM	Comment on Linguistic Form/Meaning	N/A
CONC	Concluding	Light Blue
DEF	Definitions	Yellow
DELIM	Delimiting Topic	Green
EMPH	Emphasizing	Light Gray
ENUM	Enumerating	Dark Blue
EXPL	Exemplifying with Imagining Scenarios	Brown
INTRO	Introducing the topic	Red
POST	Postponing Topic	N/A
RCAP	Recapitulating	Green
REF	Refer to Previous Idea	Blue
R&R	Repairing with Reformulating	Gray

3.1.2. Tags analysis

Table 3.2 displays if a category was identified at least once throughout all explanations. Each of these occurrences was also distributed by instructor (labeled as “I” in Table 3.2), to also be able to identify if any category is mostly used due to instructor preferences. Furthermore, we thought that it was also necessary to have data on the location of all category’s usages. So, we sorted each occurrence per one of three metrics: Beginning, Middle, and End (see Table 3.3). *Beginning* refers to the initial part of the explanation, including the introduction and additional information explained before going through each instruction. *Middle* usually means the explanation of each instruction until the instructors start to finalize the explanation. Finally, *End* includes the final parts of the explanation such as the conclusion and recapitulation of cycles and the like.

Closely observing both Tables 3.2 and 3.3, it seems that there are prominent categories among all explanations. Observing CLAR and R&R instances, it appeared that the two seemed to express the same idea, both applying when an instructor rephrased certain ideas for example when variables were explained in many cases instructors would retry to convey the general purpose of it to dissipate any possible doubts. Additionally, as the instructors were told to explain a method without any previous knowledge, this resulted in various mistakes from the fast-thinking performed at that moment. Causing several to rethink and retry to convey to further clarify each concept. As there are many similarities

Table 3.2 - Metadiscourse patterns identified from each explanation by each instructor. 4 explained 7 different methods. 2 explained 6 methods.

TAGS/Instructors	I1	I2	I3	I4	I5	I6	Total
Clarifying & R&R (CLAR & R&R)	7	7	7	7	6	5	39/40
Example (EXPL)	7	5	7	7	6	6	38/40
Introduction (INTRO)	7	6	7	6	6	6	38/40
Delimitation (DELIM)	6	7	7	5	6	5	36/40
Conclusion (CONC)	7	5	7	5	5	6	35/40
Enumeration (ENUM)	7	1	1	4	0	2	15/40
Refer to Previous Idea (REF)	0	3	4	3	2	3	15/40
Recapitulation (RECAP)	2	0	4	2	2	3	13/40
Emphasizing (EMPH)	2	2	3	2	2	1	12/40
Definition (DEF)	1	1	2	2	1	2	9/40

Table 3.3 - Metadiscourse patterns instances per location.

TAGS/Location	Beginning	Middle	End	Total
CLAR/R&R	27	58	9	84
EXPL	25	31	6	62
DELIM	46	0	0	46
INTRO	38	0	0	38
CONC	0	0	35	35
ENUM	15	6	0	21
REF	1	8	9	18
RCAP	1	9	6	16
EMPH	6	6	1	13
DEF	10	2	0	12

between these categories, for a simpler analytical perspective, we annotated them under a new category resulting from the conjunction of the previous two. Together these hold the most occurrences of a category although their location usage usually varies a lot, their occurrences can be found throughout any part of the method, especially in the middle section.

INTRO, CONC, and DELIM are the most consistent in terms of structure and location. INTRO, always being at the beginning, most of the time comprises of a simple explanation of basic objectives or features of a method. Features signifying a short description of what type of value and variable will be returned and what kind of parameters does the method receive. Although only once, a teacher simply started to go through the method development, without referring the objective, essentially skipping the beginning. Furthermore, in special cases, namely, the *Fibonacci* method, nearly all teachers felt the need to state that it is a recursive method, since this concept is one of the most probable of confuse students. The features breakdown also was normally considered DELIM segments, basing on its original definition in [30], “used to explicitly state how the topic is constrained”. Therefore, explaining the boundaries of said method (topic), what type of variable is returned, and what will be used to achieve the result such as parameters, is considered a constriction. For this reason, DELIM, are normally located inside the INTRO or in the first segments of each explanation. CONC, as a category that conveys the conclusion of each transcription, is always located on the bottom section. It is typically focused around the proceeding steps leading to the returned variable. According to our analysis, it seems to vary according to different conditions. In the sum example, various instructors opted to synthesize and explain the reasoning for the returned variable value or in the case of the *insertionSort*, an overview of the changes performed to the vector. In the *contains* example, as soon as the conditions for the boolean variable to change to true were settled, they would skip to the end and conclude with the returning of the variable. Although some instructors continued to perform a step by step analysis, normally in examples correlated to iterations of vectors. In rare cases, one of the instructors concluded an example with a real-life example, or in another singular case, there was not a conclusion, leading to an abrupt ending.

As previously stated, the previously mentioned categories were structurally consistent, but the other ones proved to not be. By going through with the analysis, these other categories although still prominent in some cases, its occurrences were rather “random”, greatly differing among instructors explaining styles. We found them more associated to certain components such as fors, whiles, ifs, and so on, of each method, rather than the location from an introduction/conclusion perspective.

DEF is not widely used among all methods transcriptions, but that seems to correlate to its usage situations. DEF was only identified in methods where instructors felt the need to further explain the definition of a concept. Namely, methods *sum*, *factorial*, and *Fibonacci*. Since these required some sort of previous mathematical knowledge or deal in harder concepts like matrices, a majority added a

simple definition of the said concept, for example, saying that a matrix is an array of arrays and following up with pictures or recapping how the Fibonacci sequence or the *factorial* functioned. This also applies in situations where the method in question was recursive. ENUM differs considerably from instructor to instructor. Instances of ENUM account mostly for when an array is involved: certain instructors, while displaying the positions to be iterated or the values of an array, do so with an enumeration. As Table 3.2 shows, instructor 1 greatly uses ENUM while others do not have the urge to do so. Meaning the usage of ENUM might not be essential for a student to understand arrays. Due to the study's context, EXPL is widely used by every instructor. Sometimes, to simply explain the functioning of some methods, a simpler example is used, one shorter to explain but still containing the core operations. As shown by Instructor 1 on the *sum* method. Another annotated usage is when real-life examples are given. Despite very rare, these offered some knowledge about the applications of said methods. For example, instructor 3 giving a playing cards example to showcase the insertion Sort method.

Regarding EMPH, its number of occurrences appear to be similar between instructors. As seen from the explanations, instructors do not seem to use it to convey the importance of the variable role. But rather, they use it in "irregular" situations in which students might fall into a mistake. As is the case of the *sum* method: *sum* contains, as a parameter, a non-square matrix, and most instructors look to rapidly say it is not a normal matrix, so additional care will be needed while iterating it. The same applies to the *max* method, to improve the efficiency of the iteration, the *max* variable is initialized with the first value of the array. So, instructors stress the reason for the benefits of doing so. Less prominent annotations account for when it is said if a variable is returned during its initialization.

Both RECAP and REF usages situations are deeply alike, both are utilized when referring to a segment or idea already explained. RECAP seems to be mainly applied in cases where certain key variables need to be closely monitored. Being the case with the *contains* method. In this example, the returned variable revolves around its boolean value, so instructors strictly remember the listener of its current value. In some other cases, it is used to remember stored values inside each array's position. REF is strictly used in situations related to repeated actions, by most instructors, as is redundant to continue to explain the same actions in a loop. Instructors sometimes start to shorten each cycle explanation, saying to repeat the same actions but not explaining them deeply as to not repeat themselves.

3.1.3. Bigram study analysis

The previous study focused on analyzing each discursive tag individually and better understand which patterns to use and where to use them. We decided to make an additional study which more

specifically examines structural relationships between tags. So, by identifying bigrams occurrences, occurrences in which occur after other tags, we can further solidify our understanding of each discursive pattern. Table 3.4 displays the obtained results, where we can observe that each bigram's occurrence is again discriminated per location of usage, using the same metrics as the previous study. Furthermore, as our objective is to identify the most prominent patterns, we decided to discard bigrams with total occurrences lesser than three.

By going through our results, we can see some obvious relationships between tags. The most prominent being an INTRO followed by a DELIM. We can notice this bigram not only has the most occurrences of all bigrams but all of those are located on the beginning part of the explanation. Meaning instructors are more likely to introduce a method and explain its boundaries such as parameters and return type. Alternatively, there are other categories that occur after an INTRO, although rarely, such as EXPL, ENUM, CLAR/R&R, and DEF that have a significant number of occurrences. Despite using the usual INTRO/DELIM beginning, sometimes instructors would use other tags instead of DELIM to adapt to certain situations of specific methods which instructors felt the need for additional explaining. Examples of this are the *insertionSort* (sorting algorithms) and the *sum* method (matrices), which, as before mentioned, might require a special usage of a category to explain certain concepts. We can also see few instances of DELIM/EXPL and DELIM/ENUM which usually occur right after an INTRO/DELIM occurrence. Both of these seem a consequence of an instructor's preference, since depending on both the level of difficulty of the method and the instructor, some choose to add an example to clarify how the method works and, in cases of an array parameter, some also enumerate each of the arrays position.

Both CLAR/R&R and EXPL hold the most occurrences among individual categories, CLAR/R&R having 84 occurrences and EXPL 62. We can see those numbers influence the results of the bigram study as both their pairs hold the second and third spots with most occurrences. Since both categories are abundantly used throughout the explanation, it is difficult to pinpoint the meaning of the usage of each pair containing either of them as these two tags not only pair up with almost every category, but their locations are used in all of the three parts of the explanation. Especially in the middle part, most bigrams with most occurrences in the middle contain either CLAR/R&R or EXPL, making it hard to draw a conclusion in bigrams revolving these two categories.

Table 3.4 - Occurrences of each bigram sorted per location.

Bigrams		Beginning	Middle	End	Total
INTRO	DELIM	35	0	0	35
CLAR/R&R	EXPL	7	15	1	23
EXPL	CLAR/R&R	7	12	0	19
INTRO	EXPL	14	0	0	14
REF	CONC	0	0	14	14
INTRO	DEF	11	0	0	11
CLAR/R&R	REF	1	8	2	11
CLAR/R&R	CONC	0	0	10	10
INTRO	CLAR/R&R	7	0	0	7
EXPL	ENUM	3	4	0	7
CLAR/R&R	EMPH	1	5	0	6
EXPL	CONC	0	0	6	6
DELIM	EXPL	6	0	0	6
ENUM	CLAR/R&R	6	0	0	6
CLAR/R&R	RCAP	0	5	0	5
EXPL	REF	0	1	4	5
INTRO	ENUM	5	0	0	5
EMPH	EXPL	2	3	0	5
EXPL	EMPH	2	2	0	4
CONC	REF	0	0	4	4
REF	EXPL	1	2	1	4
EMPH	CLAR/R&R	1	3	0	4
EXPL	RCAP	0	3	0	3
EXPL	DEF	0	3	0	3
DELIM	ENUM	3	0	0	3
REF	CLAR/R&R	0	3	0	3
RCAP	CONC	0	0	3	3

Concerning the conclusion in the explanation, we found more patterns: the cases of RCAP/CONC and REF/CONC are both used only in the end section. As previously mentioned, REF and RCAP are used to remember the student of certain quirks of a loop or variable states. As so, we can see some tendencies of instructors to do a final reviewing of the method before concluding the explanation. In some rare cases, even while concluding the explanation instructors would end with a REF, as we can see with the CONC/REF bigram. These cases occurred due to the nature of the *contains* method which highly revolved around the value of the returned variable. EMPH, its number of occurrences appear to be similar between instructors. As seen from the explanations, instructors do not seem to use it to convey the importance of the variable role. But rather, they use it in “irregular” situations in which students might fall into a mistake. As is the case of the *sum* method: *sum* contains, as a parameter, a non-square matrix, and most instructors look to rapidly say it is not a normal matrix, so additional care

will be needed while iterating it. The same applies to the *max* method, to improve the efficiency of the iteration, the *max* variable is initialized with the first value of the array. So, instructors stress the reason for the benefits of doing so. Less prominent annotations account for when it is said if a variable is returned during its initialization.

3.2. Variable roles study

To be able to examine each variable role, we used the spoken text of our previous study. First, we identified where each variable role was used throughout each method.

By going through each explanation, we manually annotated each possible pattern for each role and sorted their occurrences by location, the previously explained beginning, middle and end, and to which method component, such as *ifs* or *whiles*, the identified pattern is correlated to.

Each variable role is examined individually, and hence, the results are separated by variable role. To better align with what is going to be discussed in Chapter 4, this section uses terms of our chosen framework to refer to certain structures, respectively, Loops refer to repetition control structures such as *fors* and *whiles*, and Selections refer to decision-based structures such as *ifs*.

3.2.1. Array Index Iterator

Among all variables roles, Array Index Iterator (All) is one of the most common. This role is deeply connected to loops: we can observe most of its prominent patterns are used referring to loops properties, as it is shown in Table 3.5. The same thing applies to Stepper since an All is a specialization of it, although Steppers are used in loops that do not iterate arrays.

It can be observed that there are many ways of describing the same segment of instructions. For example, while explaining loops, some instructors proceed to use a generic short explanation of the loop, by declaring the condition for instance, "enquanto *i* não for maior que o tamanho do array". However, our most prominent Loop patterns show that instructors further simplify their explanation by specifying the loop's objective and saying that the loop will iterate certain positions of the array, which in most cases, is to iterate the whole array.

There were also some patterns in the declaration of said variable. Instructors usually opted between two different patterns. To simply say the initialized value of the variable or to be more specific and explain the objective of said initialization, meaning, its role in the loop. However, in some cases, both patterns were used in the same explanation.

Assignments revolving around an All, in this analysis, means exclusively instances where the variable is incremented or decremented. Although there are more assignments in which this role participates, as its usage is immense in variety, no patterns were discovered in these cases. So, when

the variable is incremented, similar to the last patterns, it branches to two options. When the increment is explained, instructors might say that the variable is incremented, or, in another way, say the variable goes up a value, or, by being more specific, declaring that the loop proceeds to the next position of the array.

Table 3.5 - Array Index Iterator patterns and their occurrences.

Components	Prominent Linguistic Patterns	Occurrences
Loop	To iterate all array's positions	13/22
	Declare the loop limit	9/22
	Declare the loop condition	6/22
Variable Declaration	Declare the starting value	12/22
	It starts on X position of the array	7/22
Assignments	It proceeds to the next position	12/22
	Variable is in/decremented	11/22

3.2.2. Gatherer

In beginner-level methods, there are many cases where the Gatherer is returned at the end of the method, making it the objective. Therefore, Gatherers requires additional attention when analyzing the introduction of the method and the return instruction.

As can be seen in Table 3.6, several patterns were identified. Starting from the introduction of the method, more than half of the instructors described the purpose of the returned variable, to properly establish the method objective, whereas others mention that an integer will be returned without further description.

Concerning the declaration of the variable, three different patterns appear. Explaining the initialization value of the variable, which is used by more than half of instructors, shows that is necessary to state simple aspects so as to not lose students in the explanation. The other two patterns are further specifications of the variable's role. Instructors felt the need to mention that the variable will accumulate values throughout the method. And although rare, some mentioned that the variable is returned in the end. In most cases if one pattern is identified, others of the same component are not. But in this case, both variable declarations were identified most of the time.

In case of Loops, most instructors follow the previously explained pattern of the Array Index Iterator, but further emphasize that, by explaining not only the iterated position but clarifying that those values will be stored. At the return instruction, half of the instructors decided to specify the returned value according to the role by saying that the result of the accumulation is returned. Instead of the generic alternative, declaring it returns the variable.

Table 3.6 - Gatherer patterns and their occurrences.

Components	Prominent Linguistic Patterns	Occurrences
Method Introduction	Returns an accumulation of values from X	7/10
Variable Declaration	This variable will store the accumulation of values	6/10
	Declare starting value	6/10
	This variable will be returned	3/10
Assignment	Variable will accumulate the position's value	8/10
Return	Returns the result of the accumulation	5/10
Loop	To iterate the array which values will be stored	4/5

3.2.3. Most Wanted Holder

Like Gatherer, Most Wanted Holder (MWH) also has several cases in which the objective of the method is the variable value. So, the same precautions will be noted as with the previous role. Table 3.7 presents the analysis of this role.

Table 3.7 - Most Wanted Holder patterns and their occurrences.

Components	Linguistic Patterns	Occurrences
Method Introduction	Return the max/min value of an array	6/6
Variable Declaration	Declare variable initial value	6/6
	This variable will store the wanted value	1/6
Loop	Iterate all positions to compare values	4/6
Selection	The position's value is higher than the current max value	6/6
Assignment	Store the new wanted value on the variable	6/6
Return	Returns the max value	4/6

When the method is introduced, every instructor defines the method objective, which is to return a variable containing a max or min value of a list type variable. During the declaration, everyone mentions the initialization value. But this pattern might be caused by a peculiarity. The MWH is initialized with the value of the first position of the array. Because of this, the instructors felt the need to explain the why of that initialization. Other than that, only one mentioned the role of the MWH variable during its declaration, probably because it was already explicitly said during the introduction.

During the Loop explanation, like with happens with the Gatherer, they mention which values are iterated and what is done with those values, which in this case, is to compare them. Concerning the selection and assignment which follow right up, every instructor used the same pattern: they ask if the current iterated value is higher than the old stored value and if so, they say that it replaces that value

with the new value. Concerning the return of the variable, more than half mentioned that it returns the most wanted value which in the case of the analyzed example is the highest value of the array.

3.2.4. One Way Flag

A One Way Flag variable, although it can be used in a variety of situations, it might not be returned. However, that is not the case with the examined method case. This allows us to be able to properly identify each pattern and cover more cases of this role.

Right at the beginning, when explaining the method objective, almost all instructors not only convey that the returned variable is a boolean, but additionally say in which conditions would the returned value be true or false. In this case, it refers to the condition of the Selection.

Methods where an OWF is returned focus on which conditions change the returned variable value. Therefore, throughout the declaration, perhaps because this type of method requires close observation on the OWF's value, every instructor strictly declares the initialization value of the One Way Flag variable. And half of them emphasize the fact that the variable will not change value until the condition is true.

Table 3.8 - One Way Flag patterns and their occurrences.

Components	Linguistic Patterns	Occurrences
Method Introduction	Returns a boolean value	5/6
	Explains condition for true	6/6
Variable Declaration	Declare variable initial value	6/6
	Will not change value until the condition is true	3/6
Loop	To iterate each position to (Selection condition)	4/6
Selection	Declare condition	6/6
Assignment	Update the variable's value	6/6
	It will not change values ever again	6/6
Return	Return the variable's boolean value	2/6
	Return true/false when condition	2/6

The explanation of the Loop continues to follow the previously observed pattern of other roles. Briefly explaining which positions would be iterated and state their role in the selection condition. In the Selection itself, the condition would be explained generically.

Concerning the assignment that changes the variable's value every instructor after stating that if the condition is true, says that the value would change to the new value. However, every single one was also inclined to further emphasize the fact the value would not change again, despite some instructors already having stated the same thing during the explanation of the declaration.

Concerning the return of the variable, instructors follow two different patterns, with half just stating that the boolean value of the variable would be returned, but another half also felt the need to reaffirm the condition for each of the returned boolean values.

3.2.5. Temporary

Our analyzed example with a variable with a Temporary role is very brief, only having three instructions. As this is a very simple case, the data does not provide enough examples to identify patterns related to loops or selections. Nevertheless, perhaps because of the simplicity of the method, it is possible to examine how the role is explained in the most basic ways as the method greatly focus on the swap of values of an array.

In this case, the instructors explained the swap and the importance of the auxiliary variable. In Table 3.9, we can see that all instructors used the same patterns. Then, start to state the necessity of an auxiliary variable and the value it will take. Then, explain the assignment of the temporary value to another variable. While we might still need a greater data set. The analyzed method shows the foundations of the temporary role, which is storing a value and use it for later purposes.

Table 3.9 - Temporary patterns and their occurrences.

Components	Linguistic Patterns	Occurrences
Variable Declaration	Declare starting value	6/6
	Explain the temporary role	6/6
Assignment	Explain which variable will take the temporary value	6/6

3.2.6. Most Recent Holder

As shown by the analysis portrayed in Table 3.10, the data gathered about the Most Recent Holder role is very scarce. Most of the identified patterns are usually generic and the analyzed example might not show all possible patterns for this role. The example not only does not return the Most Recent Holder but with its multiple loops, it overshadows the purpose of the variable. Many instructors only mention the variable at surface level without explaining its clear purpose in the method. Nevertheless, despite only having access to this spoken text example, we decided to continue this role study to at least have some information about it.

As previously mentioned, most information related to the MRH variable are surface-level generic explanations. Most of the time, instructors state the declaration of the initialized value of the variable, and while explaining the Loop they also mention the variable in the condition, and finally, they eventually mention the update of the variable's value. But as mentioned, each of these patterns is generic, none of them explains the variable role. Some other instructors do not even mention some

instances of the variable. This minority opted for an unusual explanation, which focused on explaining the method's main objective and functioning without mentioning secondary variables, which is the case of the MRH in this example.

Table 3.10 - Most Recent Holder patterns and their occurrences.

Components	Linguistic Patterns	Occurrences
Variable Declaration	Declare variable initial value	4/6
Loop	Declare condition	4/6
Assignment	Update of the variable value	4/6

3.2.7. Fixed Value

As Fixed Value is the simplest of all roles, describing variables which value does not change, except when changing internal values of objects and arrays (in that case, it can still be considered a Fixed Value). Several examples of Fixed Values are usually in the parameters, more so than any other role. Moreover, perhaps due to the simple nature, no specific role objective was found. Most explanations concerning it vary according to its particular purpose in the code. Making it hard to determine a common objective among all Fixed Values. Therefore, no discursive patterns were prominent enough to consider important.

However, it was already said that there are special cases with arrays and objects. For unique methods such as *swap*, where a parameter is a Fixed Value which is going to have its values modified and this action is aligned with the method's objective, several instructors, when trying to explain the method, mentioned how the values of the array will be changed and how is it going to affect the method. By knowing it is going to be modified, it makes it possible to, at the very least, minimally specify the purpose of this role in certain cases.

Prototype Implementation

This chapter describes the development of our pedagogical tool. As mentioned, our main objective of our tool is to display textual explanations of code instructions to students, so that by comparing them side-by-side, they might better understand the code's inner functioning or become aware of misconceptions. Furthermore, we also attempt to enhance the learning experience by identifying variable roles and use those to more accurately explain the instructions that relate to our second research question and previous study.

The prototype's main function is simple: given any kind of method and it displays both the source code and the textual explanation of it side-by-side to be able to locally associate each instruction to its corresponding explanation, as shown in Figures 4.1 and 4.2. Each explanation may vary according to various possibilities, whether it is differences in methods structuring or more specific changes like identifying certain variables roles (further explained in section 4.4.2).

```
class TestSum {

    double summation ( double [] array ){
        double sum ;
        sum = 0.0 ;
        int i ;
        i = 0 ;
        while ( i != array .length ){
            sum = sum + array [ i ] ;
            i = i + 1 ;
        }
        return sum ;
    }
}
```

Figure 4.2 - Source code of the method being translated (left side of the view).

Esta função devolve um double [sum](#) que resulta de uma acumulação de adições de elementos de vetor [array](#) (parâmetro).

A variável [sum](#) é [inicializada](#) com 0.0, esta variável vai acumular a cada iteração: o valor da posição [i](#) do vetor [array](#).

A variável [i](#) é [inicializada](#) com 0.

Este [ciclo](#) repete-se enquanto o valor de [i](#) é diferente do valor do [tamanho do vetor array](#).

Através de [i](#), o [ciclo](#) itera e acede a todas as posições de vetor [array](#).

- Em cada iteração [é adicionado à variável sum](#): o valor da posição [i](#) do vetor [array](#).
- [i é incrementado](#) para prosseguir para a próxima posição de vetor.

Após a acumulação de valores a função devolve o resultado dessa acumulação.

Figure 4.1 - Generated translation for the summation method displayed in Figure 4.1 (right side of the view).

4.1. Paddle

To be able to analyze the source code and identify the role of each variable, we decided to use a framework by the name of Paddle. Paddle is a recent under development framework developed by André Santos [35] and possesses various utilities, of which only the ones utilized for this project will be explained. One of its main functions is to go through each instruction, using visitors, of selected source code and it can store information in data structures relevant to each instance, which for this thesis, it simplifies the access to information for the analysis of the methods.

These data structures contain essential information about its corresponding part in the source code. Table 4.1 summarizes the most relevant data structures for our development. For example, an instance of an *IProcedure* encapsulates general details about the method such as return type, parameters, or local variables. *ILoops* and *ISelections*, referring to repetition control and decision type structures respectively, allows for a smooth analysis of guards. Variable expressions are designated as an instance of *IVariableExpression* and with the knowledge of their type, so we can adapt our generation according to each possible type.

Paddle also includes a visualization component designated Javardise. It displays the source code in a big size font making it easy to read and, similar to Eclipse, highlights keywords, for example, *whiles*, or *ifs*. An example of this already was showed in Figure 4.1. Javardise also possesses another useful functionality: The possibility to add links to in textual explanation in which when clicked highlights a specific part of the source code (example in Figure 4.3). We use this functionality to better some of the linguistic shortcomings of our prototype.

4.2. Variable role identification

This section will specify how is each variable role is identified. To do this, we perform a static analysis of the method and try to look for certain conditions common to a role. Previous research [7] already concluded that is it consistently possible to identify some roles and in doing so also theorized conditions for most roles. By making use of this research, we implement the already established conditions. However, there are a few differences, we are implementing the roles using Paddle, meaning we have easier access to more details. Therefore, we also add and change some conditions of our own to better identify roles.

Every implementation of a role is extended from an interface designated *IVariableRole* representing a role concept. We use its static method `match` which receives a variable (*IVariableDeclaration*), goes through a sequence of *ifs* to check each role using each correspondent *isVariableRole()* method (e.g., *Stepper.isStepper()*), which if true returns an *IVariableRole* instance containing said role, meaning that only one role can be returned. While doing this, we also take

measures to carefully organize each *if* as sub roles such as All, must be checked first before their parent role.

Each of the following sections will go through how each role is identified and what methods they possess to help on the method's analysis. However, all of them have a method designated *getExpressions()*. This method returns a list of program elements (*IProgramElement*) which usually contains instructions relevant to the said role. For example, MWH stores its *ISelection* guard, *ILoop* guard, and its update expression, in that order. Each *getExpressions()* is different for each role, they are considered important to explain or highlight the role's objective. We specifically use this together with the link capability of Javardise to highlight key expressions when necessary.

Table 4.1 - Relevant data structures of Paddle.

Data Structures	Description
<i>IProcedure</i>	Methods
<i>IProcedureCall</i>	Method calls
<i>ILoop</i>	Repetition control structures (e.g., <i>Fors</i>)
<i>ISelection</i>	Decision-based structures (e.g., <i>Iifs</i>)
<i>IReturn</i>	Return expressions
<i>IExpression</i>	Any expression (e.g., guards or assignments)
<i>IVariableAssignment</i>	Any kind of assignment to a variable
<i>IVariableExpression</i>	Any variable
<i>IArrayElement</i>	Element of an array (e.g., <i>array[0]</i>)
<i>ILiteral</i>	Literal value (e.g., 0, true)
<i>IUnaryExpression</i>	Any unary expression (e.g., ! <i>expression</i>)
<i>IType</i>	Type of variables (e.g., <i>int</i> , <i>boolean</i>)

```

int max ( int [] array ){
    int m ;
    m = 0 ;
    int i ;
    i = 0 ;
    while ( i < array .length ){
        if ( array [ i ] > m ){
            m = array [ i ] ;
        }
        i = i + 1 ;
    }
    return m ;
}

```

Figure 4.3 - - Example of Javardise's link function.

4.2.1. Gatherer

Gatherer is a variable which accumulates a series of values. All non-initial variable assignments to a variable must be accumulations. Moreover, all those assignments must have the same operator as it would not be a gathering operation if said values are being subtracted and added at the same time. We also considered adding a restriction and check if the accumulations are inside repetition control structures as most gatherers tend to accumulate values from arrays (e.g., adding all value from an array). We decided against it as to not restrict the system to only one kind of Gatherer.

By visiting all *IVariableAssignments* instances (excluding the declaration), we verify which ones have the correct target variable and if its operators are all the same. During the analysis, we also store information on the gatherer callable upon functions. *getOperation()* returns an *enum* instance signifying the accumulations operator such as ADD (addition) or MUL (multiplication) and *getAccumulationExpression()* returns what values is the Gatherer accumulating. This allows a more detailed explanation of the overall objective of the variable.

4.2.2. Fixed Value

Fixed Value is a variable that after its initialization never changes its value. So, it must only have one assignment, automatically returning false if it more than one is found. Although, there is one exception, if the Fixed Value points to an array or object and, for example, an assignment changes the arrays position values or the fields of an object, the variable is still considered a Fixed Value since the variable is still pointing to the same memory segment. Therefore, we check each assignment which target is the variable currently being analyzed and verify if assignments of the type [Variable = Anything] occur, excluding the first assignment (initialization), and if there are not occurrences the variable is considered a Fixed Value.

As we mentioned, exceptions as modifying values of an array may occur, so we provide the method *isModified()* which returns a boolean telling if the variable is an instance of an array or object which position values or fields were modified during the execution. We identify this aspect by checking if there are assignments of the type [ArrayElement = Value] and [Object.Field = Value]. This allows to properly distinguish the two types, modified and not modified, as they may have separate purposes in a method.

4.2.3. Stepper

A Stepper is a variable that goes through a predictable sequence of values. It is one of the most recognizable roles since in most loops there is an iterator variable controlling the limit which happens to be a Stepper. To be recognized as one, a variable must be an integer or a double (since other options

such as *Iterators* are not implemented within the framework) and after being initialized, every following assignment of that variable must be an increment or decrement of the same value during execution. We search each assignment and confirm all of them are of the type [TargetVariable (+ or -) = StepSize]. And if the operation or step size changes throughout the method it fails to identify the Stepper.

Since a Stepper is very common in loops, we provide methods to return the loop's boundaries. *getDirection()* returns an *enum* instance which can be INC or DEC, signifying an increment or decrement to the Stepper, respectively. *getStepSize()* returns an integer corresponding to the value being added to the Stepper. *getInitializationValue()* returns the initialization expression. *getCycleLimit()* returns an expression of the limit of the iteration, among some. *array.length* and 0 tend to be common limits and *isIteratingWholeArray()* returns *true* if the stepper can be theoretically used to iterate all positions of an array, although this only is helpful for AIs (next section), it was implemented in Stepper due to easier access of loop information. Both the two previous methods are achieved while analyzing the guard of the loop. If the loop guard is a common one such as, $[0 < array.length]$, $[0 \leq array.length - 1]$ or $[array.length - 1 \geq 0]$ (the left or right operand being the iterator initialization value), we both identify the limit and if it can iterate all positions of an array. Although this only covers some common conditions, it can be used to add more details about the purpose of the Stepper.

4.2.4. Array Index Iterator

The Array Index Iterator (AI) role is a sub-role of a Stepper, it has all the characteristics of a Stepper, but is also used to access positions of an array. To identify an AI we visit every *ArrayElement* instance and confirm if any index of that element is the possible AI which returns true if it finds a match. To know which arrays are being iterated, *getArrayVariables()* returns a list of iterated arrays, verified when checking all *ArrayElement* instances.

4.2.5. Most Wanted holder

The objective of a Most Wanted Holder (MWH) is to hold the value which meets most of a determined criteria. Most examples consist of finding the biggest or lowest value of an array. Among MWH usages there are similar aspects, first an iteration to search the value, a condition to see if the iterated value fits the criteria, and an update instruction if the condition is true. Although it is possible to not have an iteration involved. We decided to implement it in favor of most MWH examples.

We first go through each *ISelection* and search in its guard the possible MWH present and a relational operator such as, SMALLER (<) or GREATER (>). Following that, we verify if among the

Selection's children there is an expression of type [TargetVariable = ArrayElement] which is when the MWH is updated to a new value. Finally, it checks if the *Selection* is inside a Loop structure.

We also provide details on each MWH instance. *GetObjective()* returns an *enum* instance, most of the cases SMALLER or GREATER, meaning the objective is to find the lowest or highest value of a list of values, respectively. We achieve this by analyzing the condition for the update, if it is something common as [Array Value > Stored Highest Value], we safely assume the objective as GREATER or SMALLER, depending on the position of the operands. It can also return UNDEFINED, covering uncommon cases where the purpose is neither of the before mentioned. *getTargetArray()* returns the variable array which is being iterated and *getIteratorVariable()* returns the variable used to iterate the array (or *null* if there is none).

4.2.6. One Way Flag

A One Way Flag (OWF) role is a boolean variable that after being initialized, it may change its value according to certain conditions or not, and when it changes it will not change again. There can be various assignments and conditions for this change to occur as long as the value to be changed to is always the same.

Analysis wise, the OWF and Fixed Value are similar. We access instances of variable assignments of the possible OWF and verify if the new attributed value is a boolean value and if its opposite from the initialization value. Although there are exceptions, most practical OWF only changes when certain guards become *true*, so to give more context on why the change, we developed the method *getConditions()* which returns a list with each condition which is made by checking if each assignment is inside a parent structure *ISelection* (if) and storing its guard in the list as an expression.

4.3. Additional Analysis

Both this and the following section are not of variable roles, they are classes that use the same approach of analysis to search and identify relevant information about the method. We use two different classes, *FunctionClassifier* and *IRecursive* to search for two different aspects, which analyze an *IProcedure* instance referring to the whole method.

4.3.1. Function Classifier

FunctionClassifier is an analysis objectified to verify if a method is a procedure or a pure function (no side-effects). A procedure explicitly refers to a method that performs state changes in the objects/arrays previously allocated in the heap memory. Similar to the identification method in Fixed

Value's *isModified()*, we see if any internal values are changed such as fields of objects or position of arrays and declare procedure if it happens or function if it does not.

The method *getClassification()*, which returns the result of the analysis, an *enum* instance PROCEDURE or FUNCTION. And *getVariables()* and *getExpressions()*, both returns list with the variables whose internal values were altered, and the exact instruction where said alteration happened, respectively. So, students may be informed of the reason for it being a procedure or function.

4.3.2. Recursion

IRecursive analyzes if the method is recursive. And to identify that, we access each *IProcedureCall* and *IProcedureExpression* instance and see if the received *IProcedure* and the parent procedure of the visited expression (e.g., *expression.getProcedure()*) are the same. While analyzing, we insert into their respective lists both conditions of *ISelection* in case of a parent element *ISelection* and each recursive expression, callable using the respective method *getConditions()* and *getExpressions()*. Using both is helpful to identify the recursive and base case branches.

4.4. Prototype Development

The prototype's main function is to display both the source code and the explanation. This is controlled by two components: *ExplanationGenerator* and *ExplanationVisual*. The first focuses on generating the explanation, which when creating a new instance, receives an *IProcedure* as a parameter and generates an explanation for that method. We comprise our implementation in three actions, Analysis, Translation, and Text Smoothing. Analysis refers to storing any information relevant to translate some segments such as variable roles. Translation generates an explanation for each code element, and Text Smoothing fixes any inconsistencies for example marks, unnecessary spaces, and merging words into contractions. *ExplanationVisual* focuses on printing each text segment, respectively as a link or normal text, and creating the Standard Widget Window (SWT) window to display both the source code and the generation (see Figure 4.4).

First, we gather information about the roles of each variable as we described previously. This is applied to both local variables and parameters. If a role is discovered, the results are inputted to a list, which is used throughout the analysis to properly identify the role of variables in expressions. Additionally, if a given variable is a Fixed Value and a parameter, we include that information on a list of Fixed Value Parameter, a class used to more specifically know the purpose of the Fixed Value (discussed in section 4.4.2). We also find each return instruction and store them in a list, as those have a certain impact on translating some components.

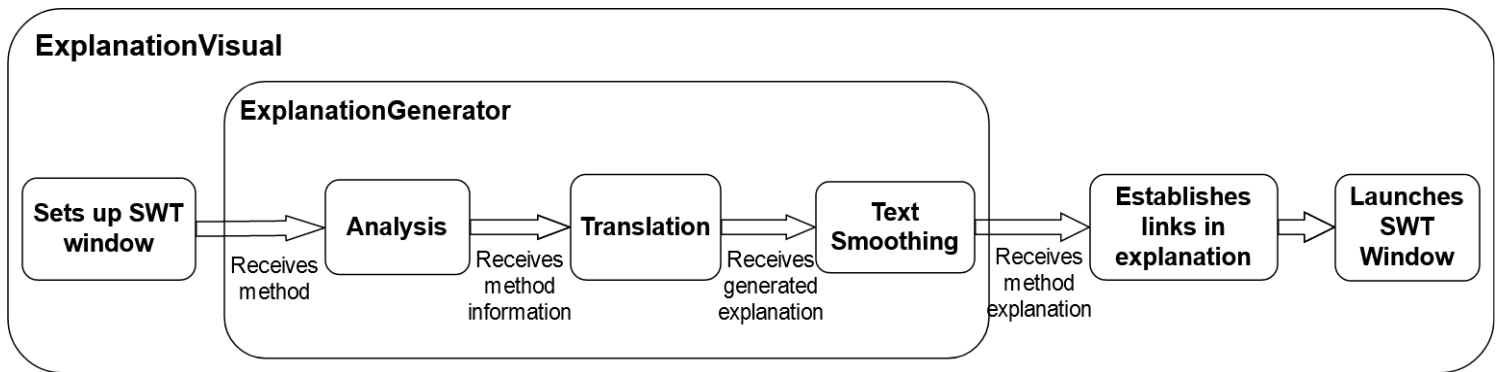


Figure 4.4 – Diagram about each step to generate an explanation.

In the initial stages of development, we had a straightforward approach visiting each instruction and directly translating each one. But the objective of this thesis is to provide a helpful explanation and enhance the generation with variable role knowledge. The results in Chapter 3 show that there are specific patterns to each role, varying in each segment of code (e.g., loops, assignments). Therefore, to achieve this variation we developed several classes to analyze each specific code segment and identify which role it is correlated to (explained in the following section). To later create the possibility of explaining certain situations in a more detailed manner.

4.4.1. Code Components

By verifying multiple examples and Paddle's capabilities we can generalize each method instruction under one of these: assignments, loops, procedure calls, returns, and selections. We created classes to specifically analyze each component. These classes were designated *AssignmentComponent*, *LoopComponent*, and so forth. So, for each one, we developed a different way to reveal its corresponding role. So, using the *getExpressions()* method available to each role instance, we compare if the program element is equal to any expression of any variable role *getExpressions()*, revealing that the component currently being analyzed correlates with that role. This is assuming that only one role can be correlated to a program element and although it is possible to have multiple roles on a single variable, considering only introductory level methods, it is quite unlikely to occur.

According to research made in Chapter 3. Some components show to be more important to some roles than others. For example: assignments, from previous results we can see that at least one assignment needs to be present in MWH, Gatherer, Stepper, and therefore All, to be identified as so. Hence, we only iterate *getExpressions()* if one of the said roles is identified. Table 4.2 shows which components are relevant to each role. Although for Loops our thought process deciding the roles was different. We saw that loops are present in many of the researched roles, but most of its discursive

patterns focused on explaining what is going to be iterated. And for that, we only need the information on the Stepper or All. Procedural calls did not reveal any relations to a specific role.

Table 4.2 – Roles related to each component.

Component	Corresponding roles
<i>Assignment</i>	MWH, Stepper, All, Gatherer
<i>Loop</i>	Stepper, All
<i>Selection</i>	OWF, MWH
<i>Return</i>	All roles
<i>ProcedureCall</i>	N/A

However, there are other possibilities to be considered. One is if what is being returned is a single variable and if that variable holds a role. If not a single variable, we cannot consistently assume a correlation between the instruction and a role, so we restrict it to returns of single variables and only then we find out if the returned variable has a role.

There is also one more component, but this one does not share the concept as the others previously mentioned, since it does not correspond to a specific segment of source code. We designated it MethodComponent, a component that stores general information about the method such as parameters or whether it is recursive, a function, or a procedure. All previous components receive access to this one since they require it to translate when it is required to mention the method type (function or procedure).

Overviewing the process, ExplanationGenerator goes through each instruction and creates a component for it which then is included in a list. However, this list only contains instructions of the same branch level, meaning instructions inside branches are not included. For branch components namely, loops and selections, they have their list to insert their components. When one of these components is created, they visit the instructions inside their branch and make components for those. Additionally, selection components have two components lists in the case of an alternative branch.

4.4.2. FVParameterComponent

On another note, for particular cases where a parameter is a Fixed Value, specific classes designated *FixedValueObjectiveVisitor* and *FVParameterComponent* are used to analyze it and try to figure out its purpose in the method. As already mentioned, Fixed Value, among all roles, does not have a clear purpose due to the simplicity of its conditions, at least in non-modified cases. However, since it is hard to pinpoint the objective without any context, another approach was taken. Instead of trying to blindly find it, we pre-define certain objectives and see if the variable is doing any of those. In basic methods,

constant parameters are commonly used to define an array's length. FixedValueObjectiveVisitor is a visitor whose only purpose is to find out if a parameter is defining an array's length. It does so by visiting every instance of array initializations (IArrayAllocation) and checking if the argument for the length matches the variables, including cases with matrices. Any results are stored in an FVParameterComponent, which can later be accessed under the form of booleans.

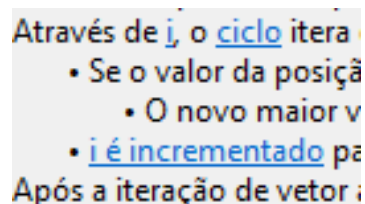
4.4.3. Method Translation

After all components are stored, we go through each one and translate those to generate an explanation. We do not store the whole translation in a single string, instead, each text segment of each line is stored in a *TextComponent*, a class we use to differentiate links from normal text. The construction of a *TextComponent* can receive either only a string or a string and an IProgramElement list. If it only receives a string it is considered normal text and vice-versa if it includes a list, containing the instructions of which the link will highlight. Every *TextComponent* is inserted into a list which then is added to another list to later be iterated and the explanation printed.

The next following sections show how each component is translated. But, before that, we describe some other important aspects for the generation.

4.4.3.1. Bullet Points

Since branches are quite common as far as simple methods go. We developed a way to add bullet points before each line to be able to differentiate in what branch is the translation included in. We achieve this by using a global integer variable designated depthLevel. Each time it goes into a branch to translate component, which can only happen on selections and loops instances, it adds to its value and removes 1 whenever it finishes translating a branch. In each translation for an instruction before any translation is done (meaning the *TextComponent* list is empty) ,depending on the depthLevel value, it adds *TextComponents* with backspaces and adds a bullet point at the end marking the beginning of the instructions (as shown in Figure 4.5).

A screenshot of text with bullet points. The text is: "Através de j, o ciclo itera". Below this are three bullet points: "• Se o valor da posiçã", "• O novo maior v", and "• j é incrementado pa". Below the bullet points is the text "Após a iteração de vetor i".

```
Através de j, o ciclo itera
• Se o valor da posiçã
• O novo maior v
• j é incrementado pa
Após a iteração de vetor i
```

Figure 4.5 – Example of bullet points before each line.

4.4.3.2. Expression Translator

Since we are working with Paddle and at the same time translating methods. We expected we would need a way to translate each code element whether it is arrays, booleans, or binary expressions. Therefore, we developed an ExpressionTranslatorPT class which contains methods to translate most of the elements in the Paddle's system. This class receives a list of *TextComponents* as a parameter which is where each method is going to input their translations. Table 4.3 shows an example of most of our translations. Although many of them follow the displayed pattern some have detailed variations depending on the circumstances.

On array elements, we felt a need to properly distinguish a vector from a matrix: translating a matrix element by only giving the positions such as "o valor da posição i,j..." may not help students to understand the specifications of a matrix. Instead, we opted to explain that the first index concerns the "rows" and the second index refers to each position inside a "row". In cases of referring to initial positions, instead of a normal translation, it becomes "o valor da primeira posição..." or in case of a matrix "o valor da primeira posição da primeira linha...". This conveys the fact that index 0 refers to the first position of an array. This also applies to the *ArrayLength* translation, for example, "o valor do tamanho da primeira linha...".

Table 4.3 - Example of a translation for each element.

Element	Example	Translation
ArrayElement	array[i]	"o valor da posição i do vetor array"
ArrayElement (matrix)	m[i,j]	"o valor da posição j da linha i da matriz m"
ArrayLength	array.length	"o valor do tamanho do vetor array"
Type	integer; boolean	"inteiro"; "booleano";
BinaryExpression	i + 1	"o valor de i a somar com 1"
UnaryExpression	!(i > 0)	"o valor de i é menor ou igual a 0"
RecordFieldExpression	o.field	"o valor do campo field do objecto o"
ProcedureCall	call(n-1)	"A função invoca o método call com os argumentos: o valor de n a subtrair por 1"
Direction (Stepper)	INC; DEC;	"incrementado"; "decrementado";
Literal	true	"verdadeiro"
VariableExpression	i	"o valor de i"
Operator	SMALLER (<)	"menor que"
Objective (MWH)	GREATER	"maior"
Assignment	i = 0	"o valor de i é alterado para 0"

As already shown, translations for basic elements storing a value such as array elements and variables, usually start with "o valor" since we wanted clear any possible misunderstandings, to make explicit that is the value of the variable being used in operations.

For types, in some particular situations involving matrices, we needed sometimes to specify the type of the elements stored in the matrices, and in others we did not. So, the `IType` method receives a boolean parameter *listsWithType* to decide whether or not to include those details. For example, "a matriz de inteiros" and "a matriz", for true and false values, of this parameter respectively.

Procedure calls follow a common pattern as shown in Table 4.3 but have a special case to consider, namely recursion. When mentioning recursion, it needs to properly convey the idea of the method calling itself. Furthermore, as far as language goes, we also wanted to depict the fact the recursion is based around a variable, meaning the results of each call will be different, hence not infinite. Two options were considered whether or not the call is in a binary expression. The grammar slightly changes for each one, for example $[i + factorial(n)]$ would be "... i mais o resultado da invocação recursiva da função...", focusing on the operation. However, if it is not in a binary expression, we can eliminate other factors and focus on explaining the recursion. For example, a recursive return such as $[factorial(n-i)]$ would be "a função é sucessivamente invocada com os argumentos n-i", but when dealing with simple arguments such as $[n-1$ or $n+1]$, it becomes "... invocada com um valor superior a n", further conveying the progress of the recursion around the variable.

Although assignments have variations for declaration and not, besides the initial segment, both follow similar rules. When dealing with primitive assignments such as integers or booleans, then it follows the patterns displayed in Table 4.3. However, for arrays and matrices, after saying that the array is assigned, it is stated its size (arrays) or "number of lines" and "lines size" (matrices). For example, the translation of the assignment `[int[][] m = new int[n][n]]` would be "A variável m é inicializada como uma matriz com o número de linhas e comprimento das mesmas igual ao valor de n (elementos a 0)". For initializations specifically, we state the starting value of each the array like the previous example "(elementos a 0)". We do the same for booleans and doubles, but change the initial value to false and 0.0, respectively.

Unary expressions "reverse" the meaning of the internal expression. There are two unary operators, NOT (!) and MINUS (-), the first is used in conditional expressions and the second in arithmetical expressions. So, whenever one of these is found, we add or modify the translations to adapt to each. For the elements that could be negated we developed a possible negated translation. In MINUS case, we considered three situations, a variable, a literal number, and a binary expression. For literals, we simply add "menos" before the number (e.g., "menos 1"). For variables, to follow our pattern of focusing on the value of the variable, we modify the text to "o valor negativo de i". Binary expressions also add "a forma negativa da expression [expression]". We considered to translate to the result of the negation instead of our approach, however, this might have caused a discrepancy between the observed source code and the translation, so we took the safer approach.

In the NOT cases, we did not do the same. In all conditional expressions where negation is found, all values are translated to its negated version. Booleans switch between "verdadeiro" e "falso" and operators such as OR (||) or AND (&&) switch between themselves. For example, `!(i < v.length)` would be translated to "o valor de i maior ou igual ao tamanho do array v".

As shown by most examples by now, binary expressions have a three step translation: translate the left operand, the operator, and the right operand. However, with commonly used expressions, we changed the approach to better explain some cases. Specifically, `[array.length - 1]` and `[array.length - 2]`, both referring to particular positions of an array, the last and penultimate. When these expressions are identified, they are directly translated to "a última posição do vetor" e "penúltima posição do vetor".

For better efficiency, we made a method for translating *Expression* instances that identifies what specific expression is used and then translates it with the appropriate approach. As all methods except this one are package-private, this method is called numerously by the components to be explained in the next sections.

4.4.3.3. Method Translator

As with many good explanations, the opening statement is important to describe the objective of the method and not confuse the students. Our translation of the method component attempts to do exactly that: it starts with an opening paragraph about its parameters, and if necessary situational information about recursion and procedures. Structurally speaking, our translation is divided into two sections, the first focusing on the declaration and the second one, being situational, adds additional information in case of recursion. Each section is represented in a different line.

The first section is the opening statement for the explanation. In section 3, we analyzed that instructor's first utterances generally explain the objective and briefly present the parameters. Our first section attempts to emulate that as much as possible. We divide this first line into three parts, each part varying according to several possibilities.

In the first part, the beginning of the sentence, usually starts with "Esta função", but in the case of a procedure, we wanted to better convey why was it considered a procedure and, as before explained, it is only a procedure when there are possible side-effects. So, the first part becomes, for example, "Este procedimento altera o vetor array passado por referência", mentioning which array/object was modified. Additionally, we add a link to "altera" that highlights which instructions modify the variable to be able to emphasize the connection between the two.

The second part focuses on what it is returned. It tries to convey the objective by looking at the return expression. To achieve this, we considered some possibilities: when there is one return and it returns a single variable or when there are two returns of either primitive boolean values or variable expressions. The first possibility varies according to the role of the returned variable. Since some roles are very specific in their usage, we take advantage of that to give an overview of the method objective. For example, a *max* function to discover the highest value of an array would be "devolve um inteiro max que contém o maior valor das posições iteradas do vetor array". By returning an MWH, we can assume the objective of the method and add details of which array is being iterated. In the same sense, we implemented a variation for each role. Table 4.4 displays each of these variations. The second possibility, of having two returns, it is explained differently: when two variables are returned the second part generally becomes "devolve var1 ou var2". We initially considered to find the role of each variable and try to do something similar to the single variable situation, but it introduced too many possibilities to be able to accurately explain the objective of the method. On the other hand, when two primitive boolean values are returned, we noticed in section 3 that instructors quickly state the conditions for returning *true* or *false*. Therefore, we adapted our generator to conceive a similar segment. We access the return, verify its condition and use that condition in the generation: for example, "devolve verdadeiro se i é igual a zero".

The third part centers around the parameters. It states their type and name, although it is basic information, it is still necessary to not make our explanation feel incomplete. Most of the time, the final part will be for example "recebe um inteiro i e um vetor v". But there is an exception: most of the time parameters will have a role of Fixed Value. However, Fixed Value does not have a clear objective, unlike other roles. But checking on the FVParameter instances, we can assert if it has a clear purpose, and if so, we attach that information to the said parameter (e.g., "recebe um inteiro n que define o comprimento do vetor v").

The second section, as stated before, is situational. It covers for cases when the method is recursive and to do this, we add another line to our opening paragraph explaining what and why it is considered recursive (e.g., "Esta função é recursiva porque se invoca a si mesma durante a sua execução"). Furthermore, to clarify any misunderstandings we add a link to "se invoca" highlighting the recursion instructions.

4.4.3.4. **Loop Translator**

Loops are useful to repeat instructions. In several cases, they are used to iterate through arrays. While explaining the loop, we want to not only translate the condition but also specifically say what that loop is used for, as accurately as possible. We divide our translation into two lines.

The first line is a basic translation of the while condition. Our objective is to specify the boundaries according to the iterator role but since there are cases that do not require an iterator, we include a default first line to minimally explain the loop. For example, a Loop with a condition $[i < v.length]$ would be translated to "Este ciclo repete-se enquanto o valor de i é menor que o valor do tamanho do vetor v ". We realized that saying "this cycle" would be too vague so we add a link to "ciclo" to highlight the referred loop. The second line tries to convey the details of the loop and it does this by first checking the role of the iterator, previously identified in the Loop Component. For each one of the two possibilities, Stepper or All, it varies the explanation since both can be used in different circumstances.

When the iterator is a Stepper, it signifies two distinct facts. The loop is controlled by the Stepper, meaning it will continue to iterate until the Stepper reached a certain value and that Stepper will continuously be incremented or decremented while the loop condition is true. We focus on conveying these two points. Using the previous example, it would generate "As iterações deste ciclo são controladas através da variável i que é incrementada 1 por 1 em cada iteração". As previously stated, "ciclo" is also linked to its respective loop and "são controladas através da variável i " highlights the condition of the guard that contains the Stepper.

Table 4.4 - An example of each implemented variation for the first statement.

Variation	Example Translation
Default	"devolve um inteiro e recebe um inteiro n "
Gatherer	"devolve um double que resulta de uma acumulação de elementos do vetor array"
Most Wanted Holder	"devolve um inteiro que tem o valor mais alto do vetor array"
Fixed Value (modified)	"devolve um vetor de inteiros array que é criado pela função e recebe um inteiro n "
One Way Flag	"devolve um booleano cujo valor vai depender de certas condições"
Stepper	"devolve um inteiro que contém o número de ocorrências de uma determinada condição"
Procedure	"altera o vetor de inteiros array passado por referência, não devolve nada e recebe um inteiro n "
2 Variables returns	"devolve o valor de inteiro a ou o valor de inteiro b "
2 literal boolean returns	"devolve verdadeiro se o valor de n resto da divisão inteira o valor de i é igual a 0 e recebe um inteiro n "

An All has more of a specific approach, we know it is used to iterate positions of an array. Therefore, we focus on explaining which positions are iterated with the variable. To cover the cases where not all the positions are iterated, we use a generic method to convey that. For example, an

iteration from the second position to the fourth would be "Através de i, o ciclo itera e acede às posições de vetor v desde a posição 2 até à posição 4". There are also situational variations that make the explanation more complete. Just in case of the variable being incremented by more than 1, it adds a short phrase describing this like such "...posição 4 a cada 2 posições", meaning it only for through half of the positions.

4.4.3.5. Selection Translator

When developing a translation for Selections, Chapter 3 results showed us that in most situations there are specific linguistic patterns. Several instructors just would state the condition of the selection and do not add any extra information according to the roles of variables, except for the MWH, which instructors would specifically refer to the old wanted value and newest possibly wanted value to update when referring to the condition. Although some patterns were identified with a OWF, since the role's main function is simple (only changing value once) and we already explain that on the variable initialization, to avoid repetition, OWF does not have an implementation.

Therefore, we developed two possible outcomes when translating a Selection. First, a generic way of stating the condition. We use *ExpressionTranslatorPT* to translate the condition. For example, an $[i!=0]$ condition would generate "Se i é diferente de 0". And in particular, for MWH cases, we adopt the same previously analyzed pattern of referring to the operands of the condition by stating the condition only happens if the newest value is higher/lower than the old value, which is stored in the MWH. So a generation of a condition $[v[i] > m]$ would turn to be as such: "Se o valor da posição do vetor v é maior que o maior valor encontrado até ao momento, guardado na variável m". We try to give more details as to why the MWH is present on the condition. Additionally, we add a link to help on the understanding of our explanation: "é maior que o ao momento" links to the condition of the *Selection*.

As far as we know, through Chapter 3 no significant patterns were noticed in the case of alternative branches so those are normally translated.

4.4.3.6. Assignment Translator

For assignments, two situations are considered: initializations (the first assignment) and the normal assignments. Initializations are when variables are introduced and so we take the advantage of enhancing that introduction. Therefore, after stating the initialized value we follow up the explanation by trying to convey the purpose of the variable with a short text according to its role.

The initialization segment is translated by *ExpressionTranslatorPT* using the already explained *translateDeclarationAssignment()* method. The short text about the role is generated by the

VariableRoleExplainer class and method *getRoleExplanationPT()* which receives a role and returns the corresponding short text. We conceived a text for each previously analyzed role focusing on the main points of their purpose. Table 4.5 displays an example of each role. Additionally, there some texts with links to highlights key instructions namely the Fixed Value, highlighting the modified values instructions, and All, showing which instructions access the array values. A full translation of a declaration, for example for an initialization [m = 0], m being an MWH, would be "A variável m é inicializada com 0, esta variável vai guardar o valor mais alto encontrado durante a iteração ao vetor m".

For the normal assignments, five variations were implemented, four that adapted to specific roles, and a generic one for cases which it is not related to any of those four. Similarly, to the declaration, the generic approach is handled by ExpressionTranslatorPT directly translating the assignment as it is in previous examples. But for the four roles, we conceived a detailed explanation based on the circumstances of each one. These roles are the MWH, the Gatherer, the Stepper, and the All. There were other considered, namely Fixed Value, however we focus on these four to first assert if providing a specific explanation is beneficial or not for the reader. Table 4.6 shows each one of those four variations. Similar to previous translations, we add links to each variation, highlighting their respective assignment.

Table 4.5 - Examples of short explanations about each role.

Role	Example Short Text
Most Wanted Holder	"esta variável vai guardar o maior/menor valor encontrado durante a iteração ao vetor m"
Gatherer	"esta variável vai acumular a cada iteração: [valor que acumula]"
Fixed Value	"após receber o seu valor, o valor desta variável nunca será alterado"
Fixed Value (Modified)	"após a sua inicialização, os valores do vetor serão alterados"
Stepper	"esta variável vai ser incrementada 1 a 1"
Array Index Iterator	"esta variável é usada para iterar e aceder a cada 2 posições do vetor v"
One Way Flag	"se o valor desta variável é alterado após essa atribuição esse nunca mais irá mudar"

4.4.3.7. Return Translator

When we analyzed the patterns discussing returns, we observed that some explanations just state what it is returned without any complementary information. However, in some cases, instructors, when mentioning the returned result, focus on the objective of the variable and remember the listener of its purpose. We focus our implementation around that thought process: we have a generic implementation to handle all non-specific cases in which we use *ExpressionTranslatorPT* to translate

the return expression. Similar to assignments, we conceived a variation for each of the following five roles: MWH, Gatherer, Stepper, OWF, and Fixed Value (modified). Furthermore, we also implemented variations for special cases such as being a recursive return or a return with no expression (void method).

Table 4.6 - Examples of assignment translations for each variation.

Variation	Assignment Translation
Default	"O valor de i é alterado para 1"
Most Wanted Holder	"O novo valor mais alto é guardado em m"
Gatherer	"Em cada iteração é adicionado à variável sum: [valor que acumula]"
Stepper	"i é incrementado por 1"
Array Index Iterator	" i é incrementado para prosseguir para a próxima posição do vetor"

Table 4.7 shows an example of every of our implemented situations. For identified role situations, we try to give a short description of what the variable contains as a value, or in case of Fixed Values, state what changes the variable went through. Since this is a return, we reinforce the idea of the purpose of the variable being correlated with the objective of the method, making it easier to understand. For void methods, since returns have no expression, hence being harder to correlate to a specific objective, we simply translate it to mark a possible ending of the corresponding method. As with other translations, we use links to once again highlight instructions of importance to the return, such as Fixed Value modification instructions.

For recursive methods, we translate the expression using *ExpressionTranslatorPT*, similar to our default approach. However, when the method is identified as recursive, for each return translation, we check if the return addresses a base case, meaning usually returns of a primitive value to finish the recursion. We do so by comparing if the return expression is included on the *IRecursion getExpressions()* list and if so, it is a recursive case. After the analysis, a tag is added right after the normal translation revealing to the reader which case is the return (e.g., "(Caso Base)", "(Caso Recursivo)"). This tag is accompanied by a link highlight the whole branch to further distinguish base branches from recursive branches.

Table 4.7 - Examples of return translations for each variation.

Variation	Return Translation Example
Most Wanted Holder	"Após a iteração de vetor a função devolve max que contém maior valor encontrado"
Stepper/Gatherer	"Após a acumulação de valores a função devolve o resultado dessa acumulação"
Fixed Value (modified)	"A função devolve vetor v cujos valores foram alterados"
Recursion (Base Case)	"A função devolve 1 (Caso Base)"
Recursion (Recursive Case)	"A função devolve o valor de n a multiplicar por o resultado da invocação recursiva desta função de n - 1 (Caso Recursivo)"
Void Method	"A função pára aqui a sua execução"

4.4.4. Text Smoothing

Throughout our previous steps, we focused on explaining to understand the method. Therefore, we did not take further care in fixing any grammar or punctuation problems as that would increase the complicity of that part of the translation process. So, we developed this specialized class to fix any text inconsistencies. It receives a list of lists of *TextComponents* from *ExplanationVisual* and it modifies that list to be displayed beside the source code.

After some observation, we took notice of these four grammar/punctuation problems: unnecessary spaces, first letter of sentences in lower case, pairs of words that could be merged into conjunctions, and missing end marks at the end of sentences. Most of these were dealt with using regular expressions, however since we use *TextComponents*, we had to adapt to the modification to other situations.

To fix these problems, we iterate through all *TextComponents* and analyze if any of them contains a grammar problem with regular expressions. To add an end mark at the end of a sentence, we simply created a new *TextComponent* containing a single mark and put it in last in each row (list). We took additional care in not adding end marks when the sentence ended with a special character such as ":". For unnecessary spacing, we identified in which cases it would happen and deleted them with regular expressions. In general, it would sometimes occur before a comma and before the end mark, with the second occurrence being caused by our solution to our first solved problem. So, using both these regexes [" *,"] and ["[]*\\$"], we replaced any matches with an empty string ("") to properly dispose of them, the second regex being applied before adding the end mark.

To turn the first letter to upper case, we had to be able to identify which *TextComponent* contained the first word since the first components can contain the code bullet points. After localizing the right component, it was also necessary to confirm if the first word was not a variable, since variables are not

normally designated with the first letter being uppercase. Only after confirming none of the previous cases was true the first letter would be modified. As already mentioned, our attention to grammar was not our focus while translating, so there are many cases in which it is possible to refer both to a male and female word. For example, our most prominent situation is when referring to an array which could be a simple array or a 2-dimensional array known as a matrix, both with different gender designations, in the Portuguese language. When generating most cases of this would be "de o", "de a", "por o", "por a". Fortunately, these are pairs that can be merged into a single word, namely contractions. To do this, we investigated most types of usual contractions [33] [34], and searched for occurrences of each combination, including a special situation where the two words are located in different *TextComponents*, and merge them into each respective combination result.

Prototype Evaluation

5.1. Preparation

After developing our prototype to be able to deal with a large range of method examples, we decided to do an initial evaluation of it and see how it fares in the eyes of people with experience not only in programming, but also in teaching the same concepts addressed in our prototype. To achieve this, we sent a questionnaire to several programming teachers and analyzed the results to see in which aspects of our prototype were still lacking and how it could be further improved.

5.2. Questionnaire Structure

We decided to build the form using Google Forms for its easy accessibility for both the creator and the people who answer it and for the several tools at the creators' disposal.

The questionnaire's structure is composed of several sections. The first section works as an introduction for the teachers to understand or know relevant information about this evaluation. More specifically, we shortly explain what the objective of the questionnaire is and for what purpose is the data going to be used. Additionally, we provide disclosure of each section and the average time for answering all sections, so the teachers know what to expect. Also, intending to maintain the results as truthful as possible we avoided mentioning the fact our textual explanations are automatically generated since this might influence the results.

The structure of each section after the first is very similar. Each one initially shows two images (as Figure 5.1 shows): a method example written in Java and its corresponding generated explanation. To purely evaluate our textual information, we provided only the generated text omitting the links to see how necessary they are to convey certain aspects such as loops. We decided to center our evaluation around three different components, Functionality, Completeness, and Readability. Functionality measures if each individual expression correctly reflects what is happening in the method. Completeness asks if relevant information was not mentioned, and Readability evaluates how easy is to read the explanation or how much the structure of our explanation holds back the conveyance of the method. We ask to evaluate each component in a scale of 1 to 5, 1 meaning a poor performance, and 5 a good one. Additionally, we also provide an open text box following each evaluation to further explain the reason for the evaluation. We repeat this structure for each of the provided examples (see appendix A).

For choosing the methods presented in the questionnaire, we opted for methods that either display concepts essential for introductory programming or methods which functioning highly revolves around a variable with a role. Table 5.1 lists all the methods used in this questionnaire and the reason why it was chosen.

1º Exemplo

```
double summation(double[] array) {
    double sum = 0.0;
    int i = 0;
    while(i != array.length) {
        sum = sum + array[i];
        i = i + 1;
    }
    return sum;
}
```

Explicação do 1º Exemplo

Esta função devolve um double que resulta de uma acumulação de elementos do vetor array (parâmetro).

A variável sum é inicializada com 0.0, esta variável vai acumular a cada iteração: o valor da posição i do vetor array.

A variável i é inicializada com 0.

Este ciclo repete-se enquanto o valor de i é diferente do valor do tamanho do vetor array.

Através de i, o ciclo itera e acede às posições do vetor array desde da posição 0 até à última posição do vetor.

- Em cada iteração é adicionado à variável sum: o valor da posição i do vetor array.
- i é incrementado para prosseguir para a próxima posição do vetor.

Após a acumulação de valores a função devolve o resultado dessa acumulação.

Figure 5.1 - Example and explanation showcase in questionnaire.

Table 5.1 - Each method provided in the questionnaire and the addressed concept.

Method	Related concept/role
<i>summation(double[]): double</i>	Gatherer (sum)
<i>max(int[]): int</i>	Most Wanted Holder (max)
<i>factorial(int): int</i>	Recursion
<i>isPrime(int): boolean</i>	Returns mid-loop
<i>naturals(int): int[]</i>	Stepper (i); Local array
<i>multiplyArrayValues(int[], int): void</i>	Procedure
<i>exists(int[], int): boolean</i>	One Way Flag (found)
<i>contains(int[][] , int): boolean</i>	Matrices, Loops within loops

5.3. Results

After sending the questionnaire to several programming teachers we got a total of 10 responses. Each one of the evaluations is displayed in Table 5.2. By looking through the gathered data, we can notice that in some methods there is a great disparity between evaluations. Our prototype to some level was able to succeed in conveying each instruction of some methods, as shown by the methods *summation* and *multiplyArrayValues*, whose results were the most positive. Readability-wise, it seems it did not prove to be a major obstacle for each instructor to understand the code, as these evaluations were mostly fours and fives except for *isPrime* and *contains*. Nevertheless, most instructors mentioned some changes to how some aspects were translated. For example, in some cases changing the textual translation of a binary expression like "o valor de n a subtrair por 1" para "n - 1" because it would be easier to understand. Despite having some better than average results, in other situations our prototype shows to perform poorly in all three components with marks below 3, namely, method *max*, *isPrime*, and *contains*.

Before analyzing the previously mentioned examples, we must discuss some common observations left by several instructors. Besides the first two examples, at least one or more instructors commented on the fact our prototype does not transfer the general idea of the method, meaning its performance is poor at explaining the high-level objective. As mentioned before our declaration translation tries to convey the objective or the conditions to return a value. This mostly happens when a single variable with a known role is returned, otherwise, the declaration will be a default translation of generic information which seems to fail at identifying the method's objective. This explains why both first methods did not have this problem, both *summation* and *max*'s objectives are greatly correlated to the returned variable role. *exists* shares some similarities, but at the time One Way Flag role was not implemented.

Several instructors also mentioned our explanation fails to mention external concepts which might be considered essential to explain certain methods. *factorial* and *isPrime* are examples of these situations. One deals with factorials and the other one with prime numbers which are not explained in the generated explanation and might be the reason for low evaluations in completeness. This also might justify the low evaluations for method *max*. We omitted the supposed pre-condition of the argument array only having values above 0 and our explanation did not mention that possibility and many instructors were quick to refer that the method is not 100% functional.

Generation-wise, both method *isPrime* and *contains* share similarities. Both return two literal boolean values. Our prototype in these cases attempts to explain the condition for it to return true or false. However, as several instructors commented that the opening declaration is weird or even confusing, this might not prove as useful as previously anticipated. In other comments, it also seems

the language of our explanation of the condition does not suffice to convey the objective. Furthermore, some instructors observed that some returns appear to be decontextualized which are obstacles to understanding the code. In *isPrime*'s example, since the generation did not explain the concept about prime numbers, more than half instructors decided to negatively evaluate the explanation. It is important to note that, among all categories, only one 3 mark was given, with the rest either being divided below and above 3, showing great inconsistency in the performance of our prototype. *contains* also possesses some unique details important to be discussed. Namely, the presence of matrices and multiples loops. Many instructors pointed out that there were no explanations of what a matrix is (arrays of arrays), meaning it was incomplete in the eyes of an experienced individual. Additionally, being the topic, most comments came from the only example featuring matrices, namely, *contains*. We can conclude that our explanation did not transmit the concept of iterations inside iterations and made the text confusing to understand.

Table 5.2 - Results gathered from the first questionnaire.

Methods	Evaluation	1	2	3	4	5
<i>summation</i>	Functionality			1		9
	Completeness					10
	Readability			2	6	2
<i>max</i>	Functionality		1	3	2	4
	Completeness		1	1		8
	Readability			1	3	6
<i>factorial</i>	Functionality				2	8
	Completeness		2		3	5
	Readability			1	3	6
<i>isPrime</i>	Functionality	2	4		3	1
	Completeness		2	1	3	4
	Readability		4		3	3
<i>naturals</i>	Functionality			2	2	6
	Completeness			1	3	6
	Readability			1	4	5
<i>multiplyArrayValues</i>	Functionality				3	7
	Completeness				4	6
	Readability				5	5
<i>exists</i>	Functionality			2	4	4
	Completeness			1	3	6
	Readability			2	3	5
<i>contains</i>	Functionality	1	4	3	2	
	Completeness			4	4	2
	Readability		2	3	4	1

5.4. Prototype modifications

After accessing the results of the questionnaire and seeing some segments were not as successful as others. We decided to add modifications to those parts which performed worse and those that despite not having a bad performance, still had a lot of comments suggesting beneficial changes to further improve the overall experience. First, we discuss the minor changes and those which are consequences of bad questionnaire results.

Regarding the initial statement, the one giving an overview of the objective, we made a few changes to specific role variations. For MWH, some instructors pointed out the first statement was a little misleading, as we mention the returned value would be the highest/lowest from an array, and to some cases where the array is not fully iterated our choice of words would be deceiving. So, for MWH, using the information on the All iterator (if there is one), it is verified whether the array is being fully iterated. If so, then we use the same pattern as is already implemented, and when not, it specifically mentions the wanted value is only among the iterated values.

For Gatherer, we added more details to describe each situation. Before it was mentioned the returned result would be from an accumulation of values, while technically true, questionnaire observations showed some room for improvement. Now it specifies what accumulations is the Gatherer variable accumulating, varying from five possible operations: addition, subtraction, multiplication, division, and the remainder of the division.

As for grammar problems, since many suggestions were made, we observed the most prominent ones and improved them. In the translation for assignments of the All variation, we fixed it changing "incrementado por 1" to "incrementado em 1 unidade" as it seems more correct from a Portuguese perspective. In what concerns MWH, throughout all translations, references to its objective vary between "o valor mais alto" and "o maior valor". Since it was brought to our attention that "o valor mais alto" might not be the best choice of words to describe the situation, we accordingly changed each objective translation to now consistently mention "o maior valor".

Returning to the first statement, originally, we had solely commas (no "e" conjunctions) separating each of the previously mentioned segments in section 4.4.3.3. In specific cases, at most, the first sentence would have 3 commas. For those who are trying to read the sentence fluently, they would be put off by the strange punctuation. Therefore, breaking their attention to the actual details of the method. To fix this we make sure the first instance of a comma is replaced by an "e" conjunction to make the text more readable (shown in Table 5.3). We also take measures to not overuse the conjunction only appearing at most two times, one to replace the first comma and another in case of multiple parameters. On another note, we got observations about certain cases of over-translating. Usually, all expressions were translated including even the more complex of binary expressions. For

expressions such as factorials, when translating a binary expression argument, it would make it hard and too long to comprehend. Therefore, we decided to not translate the arguments.

For the major problems, we already observed that the methods *isPrime* and *contains* were the ones that performed the worst among all categories. They share one similarity: both have the two returns variation in the first statement. And as noted earlier, they also share the same observations, the first statement is hard to comprehend making it difficult to discern the objective and decontextualized returns. When reading the translation for the last return after a loop, instructors were expecting an explanation regarding the loop ending followed by the execution of the last return. However, in their perspective, the translation “suddenly” explained the return without any association to the previous instructions, making it decontextualized. Particularly for *contains*, comments show our translations for loops inside loops are not adequate.

For our two returns variation, our objective was to provide insight about how the result was achieved, but both for readability reasons, the first generated sentence trying to explain the guard condition was too complex and confusing, making instructors confused on how the declaration was explained. For that reason, we reverted our two returns implementation to the default one. As for the decontextualized returns, when identifying multiple returns, the last return's translation is changed to, for example, "Caso a condição anterior não seja verdadeira a função acaba por devolver falso", "condição anterior" is a link which highlights the respective condition. Also, even if the two returns variation was reverted, we still wanted to address the fact several instructors complained about our *isPrime* first statement, more specifically, the grammar problems on the remainder of the division. So, the case of the operator for integer division was restructured to be more readable. Table 5.3 lists all modifications made to each element.

Table 5.3 - All changes made according to the results of the evaluation.

Modified Elements	Before modification	After modification
Method Translator (2 boolean return)	"devolve verdadeiro se o valor da posição r da linha i da matriz m é igual ao valor de e"	Reverted to default
Method Translator (MHW)	"devolve um inteiro que tem o valor mais alto do vetor array"	"devolve um inteiro max que tem o maior valor das posições iteradas do vetor array"
Method Translator (Gatherer)	"devolve um double que resulta de uma acumulação de elementos do vetor array"	"devolve um double sum que resulta de uma acumulação de adições de elementos do vetor array"
Method translator (commas)	"devolve um inteiro, recebe um double v"	"devolve um inteiro e recebe um double v"
Assignment Translator	"incrementado por 1"	"incrementado em 1 unidade"
ProcedureCalls (arguments)	"n a multiplicar por o resultado da invocação recursiva desta função de o valor de n a subtrair por 1"	"...desta função do valor de n-1"
Integer division	"o valor de n resto da divisão inteira o valor de i é igual a 0"	"o resto da divisão do valor de n pelo valor de i é igual a 0"
Multiple returns (last return)	"A função devolve falso"	"Caso a condição anterior não seja verdadeira a função acaba por devolver falso"

Conclusions

6.1. Main conclusions

In this thesis, our goal was to find whether a static explanation enhanced with variable role knowledge is useful for learning/teaching purposes. For this goal, we developed a prototype that can generate explanations for basic methods.

First, we researched the foundation of discourse based on previous experiments. We analyzed the textual speech of experienced programmers and categorized each of their patterns according to their locations. We then applied our results to build the foundation of our text structure. Using the same transcriptions, we also analyzed if each role of each variable influenced the instructors to explain them in a particular way and when so, tried to apply any of those to enhance the explanation.

We built a prototype that for generating textual explanations for an arbitrary method involving basic programming constructs. First, each variable (including parameters), is analyzed to find out what role it has in the method. Following that, it starts to visit each instruction and store its information in specific data structures, where each does its own analysis to determine a role. Secondly, it starts to translate each of those components in order, where each has variations according to the specific situation. Lastly, we clean up text inconsistencies such as grammar. After the generation, we make use of Javardise, Paddle's visual component, to create a window with both the generated explanation and the source code for a side-by-side comparison.

The results of the evaluation show that, in some methods, our prototype achieved a good performance, proving some evidence that our approach is adequate, at least from the perspective of an experienced programming instructor. However, there were also methods in which our explanation received more negative results than positive among all instructors. We analyzed the prominent problems and attempted to fix them.

To answer our research questions:

- **How are variables with different roles explained by instructors using natural language?** - By analyzing the textual transcriptions, we were able to determine that there are specific discourse patterns to describe each role, although, these patterns seem clearer in roles with an evident purpose such as the MWH and the Gatherer.
- **How helpful is the knowledge of variable roles on a static explanation using natural language?** - Depending on the role, results showed good evaluations on some methods.

However, we learned that the prototype could not capture the objective of the method, ending up explaining it vaguely. This was not the case with MWH and Gatherer. These two roles have a clear purpose and, in our evaluation, both example's objective was to return the value of the variable with such a role. Showing if certain conditions are present, variable role knowledge can be used to determine even the method's objective. On another level, role knowledge can also be used to explain common occurrences such as loops. By using the All, we can determine which arrays and what positions are being iterated, making possible a precise loop explanation. In some examples, roles allow for a more immersive explanation, which is necessary for a good learning experience.

- **Is it possible to automatically generate meaningful explanations of small functions in natural language?** – Yes, although there are exceptions, most of the translations overall received positive results, many of them having evaluation scores of 5 (the highest grade), meaning that, to some extent, instructors considered the translations adequate to convey the methods' functionality.

6.2. Future work

For future work, first, we propose to increase the data gathered about patterns of each role. Although our data set is based on previous work [17], due to the lack of data, some roles could not be analyzed as wanted. Therefore, we hope the same experiment is redone but with more instructors and methods to add more variety to the data set, consequently, uncovering more patterns and adapting those to the translations. Secondly, we also desire to expand on our evaluation method, exhibiting this prototype to more experienced instructors. And, especially, to make an experience to see how our project fares when in the hands of an inexperienced student, which is the main purpose of this prototype.

Just as this project is a continuation of PandionJ [7] in seeing how much the knowledge of roles can enhance the initial programming learning experience and what kind of portrayal (e.g., textual, visual) makes the most difference, we are encouraged to continue to experiment with more information portrayals, hopefully leading to a conclusion about the general benefits of variable roles in education.

Bibliography

- [1] Y. Qian and J. Lehman, "Students' misconceptions and other difficulties in introductory programming: A literature review," *ACM Transactions on Computing Education*, vol. 18, no. 1, pp. 1–24, 2017, doi: 10.1145/3077618.
- [2] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud, "Predicting at-risk novice Java programmers through the analysis of online protocols," *ICER'11 - Proceedings of the ACM SIGCSE 2011 International Computing Education Research Workshop*, pp. 85–92, 2011, doi: 10.1145/2016911.2016930.
- [3] 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)*, pp. 1–9, 2018, doi:10.1145/3279720.3279732
- [4] M. Kölling, "Using BlueJ to introduce programming," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4821 LNCS, pp. 98–115, 2008, doi: 10.1007/978-3-540-77934-6_9.
- [5] A. L. Santos, "AGUIA/J: A tool for interactive experimentation of objects," *ITiCSE'11 - Proceedings of the 16th Annual Conference on Innovation and Technology in Computer Science*, no. November, pp. 43–47, 2011, doi: 10.1145/1999747.1999762.
- [6] 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," *Educational Psychologist*, vol. 21, no. 41, pp. 75–86, 2006, doi: 10.1207/s15326985ep4102_1.
- [7] H. S. Sousa, "Illustrating Debugger Execution Leveraging on Variable Roles," MSc, Ista, Iscte - Instituto Universitário de Lisboa, 2016, Accessed on: Nov 23, 2019. [Online]. Available: <https://repositorio.iscte-iul.pt/handle/10071/14632>.
- [8] E. Lahtinen, K. Ala-Mutka, and H. M. Järvinen, "A study of the difficulties of novice programmers," *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pp. 14–18, 2005, doi: 10.1145/1067445.1067453.
- [9] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting Java programming errors for introductory computer science students," *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)*, pp. 153–156, 2003, doi: 10.1145/792548.611956.
- [10] P. Denny, A. Luxton-Reilly, and E. Tempero, "All syntax errors are not equal," *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, pp. 75–80, 2012, doi: 10.1145/2325296.2325318.
- [11] Simon, "Assignment and sequence: Why some students can't recognise a simple swap," *Proceedings - 11th Koli Calling International Conference on Computing Education Research, Koli Calling'11*, pp. 10–15, 2011, doi: 10.1145/2094131.2094134.

- [12] J. Sajaniemi, "An empirical analysis of roles of variables in novice-level procedural programs," *Proceedings - IEEE 2002 Symposia on Human Centric Computing Languages and Environments, HCC 2002*, pp. 37–39, 2002, doi: 10.1109/HCC.2002.1046340.
- [13] U. Nikula, J. Sajaniemi, M. Tedre, and S. Wray, "Python and Roles of Variables in Introductory Programming: Experiences from Three Educational Institutions," *Proceedings of the 2007 InSITE Conference*, vol. 6, 2007, doi: 10.28945/3097.
- [14] M. Kuittinen and J. Sajaniemi, "Teaching roles of variables in elementary programming courses," *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)*, vol. 36, no. 3, pp. 57–61, 2004, doi: 10.1145/1026487.1008014.
- [15] P. Gerdt and J. Sajaniemi, "An Approach to Automatic Detection of Variable Roles in Program Animation," *Third Program Visualization Workshop*, no. 1984, pp. 86–93, 2004.
- [16] L. N. de Barros, A. Paula, and S. P. Brazil, "A Tool for Programming Learning with," pp. 125–129, 2005.
- [17] A. L. Santos, H. Sousa, "An exploratory study of how programming instructors illustrate variables and control flow. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17)*, pp. 173–177, 2017, doi: 10.1145/3141880.3141892.
- [18] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," *ASE'10 - Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 43–52, 2010, doi: 10.1145/1858996.1859006.
- [19] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," *IEEE International Conference on Program Comprehension*, pp. 71–80, 2011, doi: 10.1109/ICPC.2011.28.
- [20] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," *Proceedings - International Conference on Software Engineering*, pp. 101–110, 2011, doi: 10.1145/1985793.1985808.
- [21] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," *IEEE International Conference on Program Comprehension*, pp. 23–32, 2013, doi: 10.1109/ICPC.2013.6613830.
- [22] E. Wong, T. Liu, and L. Tan, "CloCom: Mining existing source code for automatic comment generation," *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, pp. 380–389, 2015, doi: 10.1109/SANER.2015.7081848.
- [23] Y. Liang and K. Q. Zhu, "Automatic generation of text descriptive comments for code blocks," *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pp. 5229–5236, 2018.

- [24] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," *Proceedings - International Conference on Software Engineering*, pp. 200–210, 2018, doi: 10.1145/3196321.3196334.
- [25] D. Miguel, "Automatic Generation of Descriptions for Prolog Programs," MSc, Ista, Iscte - Instituto Universitário de Lisboa, 2019, Accessed on: Jul 13, 2020. [Online]. Available: <https://www.it.pt/Supervisions/Supervision/16115>.
- [26] N. R. Carvalho, J. J. Almeida, P. R. Henriques, and M. J. Varanda, "From source code identifiers to natural language terms," *Journal of Systems and Software*, vol. 100, pp. 117–128, 2015, doi: 10.1016/j.jss.2014.10.013.
- [27] D. K. Deeptimahanti and R. Sanyal, "Semi-automatic generation of UML models from natural language requirements," *Proceedings of the 4th India Software Engineering Conference 2011, ISEC'11*, pp. 165–174, 2011, doi: 10.1145/1953355.1953378.
- [28] S. Mandal and S. K. Naskar, "Natural Language Programing with Automatic Code Generation towards Solving Addition-Subtraction Word Problems," *Proceedings of the 14th International Conference on Natural Language Processing (IICON-2017)*, no. December, pp. 146–154, 2017, [Online]. Available: <https://www.aclweb.org/anthology/W17-7519>.
- [29] R. Correia, N. Mamede, J. Baptista, and M. Eskenazi, "MetaTED: A corpus of metadiscourse for spoken language," *Proceedings of the 10th International Conference on Language Resources and Evaluation, LREC 2016*, pp. 3907–3913, 2016.
- [30] A. Ädel, "Just to give you kind of a map of where we are going: A Taxonomy of Metadiscourse in Spoken and Written Academic English," *Nordic Journal of English Studies*, vol. 9, no. 2, p. 69, 2010, doi: 10.35360/njes.218.
- [31] R. Correia, N. Mamede, J. Baptista, and M. Eskenazi, "Using the crowd to annotate metadiscursive acts," *Proceedings 10th Joint ISO - ACL SIGSEM Workshop on Interoperable Semantic Annotation*, pp. 102–108, 2014.
- [32] Python Speech Recognition, "SpeechRecognition · PyPI," *Pypi.org*, 2019. <https://pypi.org/project/SpeechRecognition/> (accessed Jul. 13, 2020).
- [33] "Combinação e contração das preposições - PrePara ENEM." <https://www.preparaenem.com/portugues/combinacao-contracao-das-preposicoes.htm> (accessed Sep. 08, 2020).
- [34] "Contrações (preposição + pronome/determinante, pronome + pronome) - Ciberdúvidas da Língua Portuguesa." <https://ciberduvidas.iscte-iul.pt/consultorio/perguntas/contraccoes-preposicao--pronomedeterminante-pronome--pronome/27390> (accessed Sep. 19, 2020).
- [35] "GitHub - andre-santos-pt/paddle." [Online]. Available: <https://github.com/andre-santos-pt/paddle>. [Accessed: 28-Oct-2020].

Appendix A

Questionnaire used to evaluate the textual generation prototype

Questionnaire general information

Avaliação de explicações de código

Antes de tudo, muito obrigado por aceitar participar neste estudo.

O objectivo é avaliar explicações textuais de métodos escritos em Java. Os dados serão utilizados no contexto da dissertação de Mestrado em Engenharia Informática de Ricardo Silva (ISCTE-IUL), sob a orientação de André Santos e Ricardo Ribeiro.

Em cada secção será apresentado um método junto com uma explicação textual, seguido de perguntas para avaliar essa explicação.

Prevê-se que o formulário demore cerca de 30 minutos a responder.

Caso o tamanho de letra seja demasiado pequeno, pode utilizar CTRL+ para o aumentar.

Nome

(apenas para efeitos de controlo de participantes)

A sua resposta

Method 1

1º Exemplo

```
double summation(double[] array) {
    double sum = 0.0;
    int i = 0;
    while(i != array.length) {
        sum = sum + array[i];
        i = i + 1;
    }
    return sum;
}
```

Explicação do 1º Exemplo

Esta função devolve um double que resulta de uma acumulação de elementos do vetor array (parâmetro).

A variável sum é inicializada com 0.0, esta variável vai acumular a cada iteração: o valor da posição i do vetor array.

A variável i é inicializada com 0.

Este ciclo repete-se enquanto o valor de i é diferente do valor do tamanho do vetor array.

Através de i, o ciclo itera e acede às posições do vetor array desde da posição 0 até à última posição do vetor.

- Em cada iteração é adicionado à variável sum: o valor da posição i do vetor array.
- i é incrementado para prosseguir para a próxima posição do vetor.

Após a acumulação de valores a função devolve o resultado dessa acumulação.

Method 2

2º Exemplo

```
int max(int[] array) {
    int m = 0;
    int i = 0;
    while(i < array.length) {
        if(array[i] > m) {
            m = array[i];
        }
        i = i + 1;
    }
    return m;
}
```

Explicação do 2º Exemplo

Esta função devolve um inteiro que tem o valor mais alto do vetor array (parâmetro).

A variável m é inicializada com 0, esta variável vai guardar o valor mais alto de um vetor de inteiros array.

A variável i é inicializada com 0.

Este ciclo repete-se enquanto o valor de i é menor que o valor do tamanho do vetor array.

Através de i, o ciclo itera e acede às posições do vetor array desde da posição 0 até à última posição do vetor.

- Se o valor da posição do vetor array for maior que o maior valor encontrado até ao momento guardado na variável m.
- O novo valor mais alto é guardado em m.

- i é incrementado para prosseguir para a próxima posição do vetor.

Após a iteração do vetor a função devolve a variável com o valor mais alto do vetor.

Method 3

3º Exemplo

```
int factorial(int n) {
    if(n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Explicação do 3º Exemplo

Esta função devolve um inteiro, recebe um inteiro n (parâmetro).

Esta função é recursiva porque se invoca a si mesma durante a sua execução.

Se o valor de n é igual a 0.

- A função devolve 1 (Caso Base).

Caso contrário:

- A função devolve o valor de n a multiplicar pelo resultado da invocação recursiva desta função do valor de n a subtrair por 1 (Caso Recursivo).

Method 4

4º Exemplo

```
boolean isPrime(int n) {
    int i = 2;
    while(i < n) {
        if(n%i == 0) {
            return true;
        }
        i = i + 1;
    }
    return false;
}
```

Explicação do 4º Exemplo

Esta função devolve verdadeiro se o valor de n resto da divisão inteira o valor de i é igual a 0, recebe um inteiro n (parâmetro).

A variável i é inicializada com 2, esta variável vai ser incrementada 1 a 1.

Este ciclo repete-se enquanto o valor de i é menor que o valor de n.

As iterações deste ciclo são controladas através da variável i que é incrementado por 1 em cada iteração.

- Se o valor de n resto da divisão inteira o valor de i é igual a 0.
 - A função devolve verdadeiro.
- i é incrementado por 1.

A função devolve falso.

Method 5

5º Exemplo

```
int[] naturals(int n) {
    int[] array = new int[n];
    int i = 0;
    while(i < n) {
        array[i] = i + 1;
        i = i + 1;
    }
    return array;
}
```

Explicação do 5º Exemplo

Esta função devolve um vetor de inteiros que é criado pela função, recebe um inteiro n (parâmetro) que define o comprimento do vetor array.

A variável array é inicializada como um vetor com tamanho igual ao valor de n (elementos a 0), após a sua inicialização, os valores do array serão alterados.

A variável i é inicializada com 0.

Este ciclo repete-se enquanto o valor de i é menor que o valor de n.

Através de i, o ciclo itera e acede às posições do vetor array desde da posição 0 até à posição n.

- O valor da posição i do vetor array é alterado para o valor de i a somar com 1.
- i é incrementado para prosseguir para a próxima posição do vetor.

A função devolve o vetor array cujos valores foram alterados.

Method 6

6º Exemplo

```
void multiplyArrayValues(int[] array, int n) {
    int i = 0;
    while(i < array.length) {
        array[i] = array[i] * n;
        i = i + 1;
    }
}
```

Explicação do 6º Exemplo

Este procedimento altera o vetor de inteiros array passado por referência, recebe um inteiro n (parâmetro).

A variável i é inicializada com 0.

Este ciclo repete-se enquanto o valor de i é menor que o valor do tamanho do vetor array.

Através de i, o ciclo itera e acede às posições do vetor array desde da posição 0 até à última posição do vetor.

- O valor da posição i do vetor array é alterado para o valor da posição i do vetor array a multiplicar pelo valor de n.
- i é incrementado para prosseguir para a próxima posição do vetor.

7º Exemplo

```
boolean exists(int[] array, int e) {
    boolean found = false;
    int i = 0;
    while(!found && i < array.length) {
        if(array[i] == e) {
            found = true;
        }
        i = i + 1;
    }
    return found;
}
```

Explicação do 7º Exemplo

Esta função devolve um booleano, recebe um vetor de inteiros array (parâmetro) e um inteiro e (parâmetro).

A variável found é inicializada com falso.

A variável i é inicializada com 0.

Este ciclo repete-se enquanto o valor da variável found for falso e o valor de i é menor que o valor do tamanho do vetor array.

Através de i, o ciclo itera e acede às posições do vetor array desde da posição 0 até à última posição do vetor.

- Se o valor da posição i do vetor array é igual ao valor de e.
 - found é alterado para verdadeiro.
- i é incrementado para prosseguir para a próxima posição do vetor.

A função devolve o valor de found.

Method 8

8º Exemplo

```
boolean contains(int[][] m, int e) {
    int i = 0;
    while(i < m.length) {
        int r = 0;
        while(r < m[0].length) {
            if(m[i][r] == e) {
                return true;
            }
            r = r + 1;
        }
        i = i + 1;
    }
    return false;
}
```

Explicação do 8º Exemplo

Esta função devolve verdadeiro se o valor da posição r da linha i da matriz m é igual ao valor de e, recebe um matriz de inteiros m (parâmetro) e um inteiro e (parâmetro).

A variável i é inicializada com 0.

Este ciclo repete-se enquanto o valor de i é menor que o valor do número de linhas da matriz m.

Através de i, o ciclo itera e acede às posições do vetor m desde da posição 0 até à última posição do vetor.

- A variável r é inicializada com 0.

- Este ciclo repete-se enquanto o valor de i é menor que o valor do tamanho da primeira linha da matriz m.

- Através de r, o ciclo itera e acede às posições do vetor m desde da posição 0 até à última posição do vetor.

- Se o valor da posição r da linha i da matriz m é igual ao valor de e.

- A função devolve verdadeiro.

- r é incrementado para prosseguir para a próxima posição do vetor.

- i é incrementado para prosseguir para a próxima posição do vetor.

A função devolve falso.

Questions about functionality

Classifique a explicação em termos das suas afirmações refletirem corretamente o funcionamento do código. *

Muitas incorreções 1 2 3 4 5 Sem incorreções

Caso considere que alguma parte do código não tenha sido explicada corretamente, especifique na caixa de texto abaixo.

A sua resposta

Questions about completeness

Classifique a explicação em termos da sua completude (se explica tudo o que é relevante). *

Muito incompleta 1 2 3 4 5 Completa

Caso considere que alguma questão relevante está em falta, especifique na caixa de texto abaixo.

A sua resposta

Questions about readability

Classifique o texto quanto à sua legibilidade. *

	1	2	3	4	5	
Ilegível	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Perfeitamente legível

Caso tenha sugestões para melhorar a forma do texto da explicação, especifique na caixa de texto abaixo.

A sua resposta

Appendix B

Java methods analyzed in chapter 3 from [17]

Java Method 1: factorial

```
int factorial(int x) {
    int result = 1;
    int i = x;
    while (i > 1){
        result *= i;
        i--;
    }
    return result;
}
```

Java Method 2: fibonacci

```
int fibonacci(int n){
    if(n <= 1)
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Java Method 3: swap

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

Java Method 4: contains

```
boolean contains(int target, int[] array) {
    boolean exists = false;
    for(int i = 0; i < array.length; i++) {
        if(array[i] == target)
            exists = true;
    }
    return exists;
}
```

Java Method 5: max

```
int max(int[] array) {
    int max = array[0];
    for(int i = 1 ; i < array.length; i++) {
        if(array[i] > max)
            max = array[i];
    }
    return max;
}
```

Java Method 6: insertionSort

```
void insertionSort(int[] array) {
    int n = array.length;
    for(int j = 1; j < n; j++) {
        int key = array[j];
        int i = j - 1;
        while (i > -1 && array[i] > key) {
            array[i+1] = array[i];
            i--;
        }
        array[i+1] = key;
    }
}
```

Java Method 7: sum

```
int sum (int[][] array) {
    int sum = 0;
    for (int i = 0 ; i < array.length; i++) {
        for (int j = 0; j < array[i].length; j++) {
            sum += array[i][j];
        }
    }
    return sum;
}
```
