

iscte

INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Coding by voice in a projectional editor

Alexandre Filipe Magalhães Cancelinha

Master in Computer Science and Engineering

Supervisor:

Prof. Dr. André L. Santos, Assistant Professor

Iscte – Instituto Universitário de Lisboa

Co-Supervisor:

Prof. Dr. Fernando Batista, Associate Professor

Iscte – Instituto Universitário de Lisboa

October, 2023

iscte

TECNOLOGIAS
E ARQUITETURA

Department of Information Science and Technology

Coding by voice in a projectional editor

Alexandre Filipe Magalhães Cancelinha

Master in Computer Science and Engineering

Supervisor:

Prof. Dr. André L. Santos, Assistant Professor
Iscte – Instituto Universitário de Lisboa

Co-Supervisor:

Prof. Dr. Fernando Batista, Associate Professor
Iscte – Instituto Universitário de Lisboa

October, 2023

Acknowledgements

I'm extremely grateful to both of my supervisors, Professor André Santos and Professor Fernando Batista, for helping me in every way possible in order to complete this dissertation, especially when it came to writing the dissertation, this help was really invaluable.

I would also like to thank my parents, Fernando and Célia, for being able to provide all kinds of help to achieve the best possible result.

I would also like to thank my grandparents, Ana and Manuel, for helping me to relax and gain strength for the project and for always believing in me and my abilities.

I would like to thank my brothers, Rafael and Bernardo, along with my group of friends, for distracting me from problems that appeared during the course of the dissertation.

Resumo

O aumento da produção de software está associado a um aumento significativo de doenças ocupacionais, tais como Lesão por Esforço Repetitivo (RSI), com origem na repetição excessiva de movimentos em partes do corpo, como ombros, cotovelos, pulsos e mãos. Essas condições podem ter impacto negativo na produtividade dos programadores, que dependem dessas partes para realizar tarefas essenciais. Com o objetivo de abordar esse problema, foram feitos esforços para encontrar uma solução que substituísse o uso do teclado e rato por comandos de voz. Neste contexto, o Jasay foi concebido com a finalidade de combinar a ideia de um editor estruturado com a integração de comandos de voz dividindo-se em duas partes. A primeira parte envolve a captação e a transcrição de comandos de voz para texto, enquanto a segunda consiste no processamento desses comandos de forma a executar alterações ao código fonte Java.

Com vista a avaliar a eficácia deste protótipo, foram realizados dois testes. O primeiro teste envolveu a participação de utilizadores, cujos resultados foram positivos, conseguindo concluir as tarefas num tempo razoável utilizando o Jasay. O segundo teste comparou o tempo de execução de algumas tarefas entre o Jasay e a utilização do teclado e rato. Embora a combinação teclado-rato tenha apresentado um desempenho ligeiramente superior, a diferença pode ser considerada aceitável.

Este projeto representa uma abordagem inovadora na procura de soluções que visam melhorar a saúde e produtividade dos programadores, ao reduzir a dependência de movimentos repetitivos e introduzindo uma alternativa viável baseada em comandos de voz.

Palavras-chave: Programar com a voz, Editor Estruturado, Reconhecimento de Fala.

Abstract

The increase in software production is associated with a significant rise in occupational illnesses, such as Repetitive Strain Injury (RSI), which stems from the excessive repetition of movements in parts of the body such as the shoulders, elbows, wrists and hands. These conditions can have a negative impact on the productivity of programmers, who depend on these parts to carry out essential tasks. In order to address this problem, efforts have been made to find a solution that replaces the use of the keyboard and mouse with voice commands. In this context, Jasay was conceived with the aim of combining the idea of a structured editor with the integration of voice commands in two parts. The first part involves capturing and transcribing voice commands into text, while the second consists of processing these commands in order to make changes to the Java source code.

In order to assess the effectiveness of this prototype, two tests were carried out. The first test involved the participation of users, whose results were positive, managing to complete the tasks in a reasonable time using Jasay. The second test compared the execution time of some tasks between Jasay and using the keyboard and mouse. Although the keyboard-mouse combination performed slightly better, the difference can be considered acceptable.

This project represents an innovative approach in the search for solutions aimed at improving the health and productivity of programmers by reducing dependence on repetitive movements and introducing a viable alternative based on voice commands.

Keywords: Coding with voice, Projectional Editor, Speech Recognition.

Index

1	INTRODUCTION	1
1.1	GOALS	1
1.2	RESEARCH QUESTIONS	2
1.3	CONTRIBUTIONS.....	2
1.4	DISSERTATION OUTLINE.....	2
2	RELATED WORK	3
2.1	CODE EDITORS USING VOICE COMMANDS	3
2.2	SPEECH RECOGNITION SYSTEMS.....	6
2.3	PROJECTIONAL EDITORS.....	7
3	APPROACH	9
3.1	SPEECH RECOGNITION.....	9
3.2	SPEECH INTERPRETATION	10
3.2.1	<i>Command Validation</i>	11
3.2.1.1	Command grammar	11
3.2.1.2	List of possibilities on cursor position	13
3.2.1.3	Filtering procedure.....	15
3.2.2	<i>Command interpretation</i>	16
3.2.2.1	Commands for changing the class.....	17
3.2.2.2	Commands for changing the method.....	18
4	JASAY PROTOTYPE.....	21
4.1	SPEECH RECOGNITION.....	22
4.2	SPEECH INTERPRETATION	23
4.2.1	<i>JavaParser</i>	23
4.2.2	<i>Javardise</i>	24
4.2.3	<i>Command grammar</i>	24
4.2.4	<i>List of Possibilities</i>	24
4.2.5	<i>Filtering procedure</i>	25
4.2.5.1	Levenshtein distance.....	25
4.2.5.2	Phonetic Similarity	26
4.2.5.3	Camel case	26
4.2.5.4	Numbers.....	27
4.2.6	<i>Navigating the Editor</i>	28
4.2.7	<i>Status Feedback</i>	29

4.2.8	<i>Commands for changing the class</i>	30
4.2.9	<i>Commands for changing the method</i>	32
4.2.9.1	Statements.....	32
4.2.9.2	Expressions.....	36
4.2.9.3	Types.....	38
5	PERFORMANCE EVALUATION	39
5.1	USER TESTS.....	39
5.1.1	<i>Tasks definition</i>	39
5.1.2	<i>Code writing performance results</i>	43
5.1.3	<i>Analysis of the results</i>	44
5.2	JASAY VERSUS KEYBOARD AND MOUSE.....	46
5.2.1	<i>Experiments and results</i>	46
5.2.2	<i>Analysis of the results</i>	47
6.	CONCLUSIONS AND FUTURE WORK	49
6	REFERENCES	51

INDEX OF TABLES

Table 3.1. Commands allowed within the class body.	17
Table 3.2. Commands to modify statements allowed within the body of the method. STC corresponds to the space that the user can change.....	18
Table 3.3. Commands to modify expressions. STC corresponds to the space that the user can change. CV corresponds to the value where the cursor is placed.	19
Table 5.1. Average times and commands used compared to the expected times and commands for each of the tasks	43
Table 5.2. Times made with the Jasay versus time made with the keyboard and mouse.....	46
Table 5.3. Overall times made with Jasay and Keyboard and mouse.	47

INDEX OF FIGURES

Figure 2.1. Example if statement created from the voice command “if current record number is less than max offset then”	3
Figure 3.1. General overview of the system's main components.	9
Figure 3.2. General overview of the procedures in the Speech Interpretation part.	11
Figure 3.3. Grammar used in the prototype.	12
Figure 3.4. Example of when the cursor is in an access modifier.	13
Figure 3.5. Example of when the cursor is in a type of a variable.	14
Figure 3.6. Example of when the cursor is in a name of method declaration.	14
Figure 3.7. Example of when the cursor is in a statement, in this case an empty statement.	15
Figure 3.8. Example of when the cursor is on an expression. The first image shows an expression composed only of a number and the second an expression composed of an arithmetic expression.	15
Figure 4.1. General outline of Jasay implementation.	22
Figure 4.2. Example of a list of possibilities getting information from the AST.	25
Figure 4.3. Levenshtein distance between the word “while” and “wild”.	25
Figure 4.4. Words “while” and “wild” in their form of grapheme and phoneme.	26
Figure 4.5. In the first image the initial state of the cursor is in the statement type, in this case “int”, in the second image after executing the command to move the cursor to the right the cursor moves to the name of the variable, “x”.	29
Figure 4.6. Status feedback showing the recognizer listening and the last command said by the user.	30
Figure 4.7. Different status of the recognizer.	30
Figure 4.8. The structure of the field declaration with no equals and for a method.	31
Figure 4.9. Command that creates an assignment statement with value.	33
Figure 4.10. Command that creates an assignment statement just by saying the type.	33
Figure 4.11. Command that creates an assignment statement just by saying the type and name.	34
Figure 4.12. Command that creates an assignment statement with a function call.	34
Figure 4.13. Command that creates an if statement.	34
Figure 4.14. Command that creates an if statement with an expression.	35
Figure 4.15. Command that creates an else statement.	35
Figure 4.16. Command that creates a for statement.	35
Figure 4.17. Command that creates a for statement with an expression.	36

Figure 4.18. First a command that creates a while statement without an expression and below a
command that creates a while statement with an expression..... 36

Figure 4.19. Creation of a compound expression with the first value already inserted..... 37

Figure 4.20. Creation of a complete compound expression. 37

Figure 4.21. Creation of an expression with a function call inside. 37

Figure 5.1. Task 1 objective. 39

Figure 5.2. Task 2 objective. 40

Figure 5.3. Task 3 objective. 40

Figure 5.4. Task 4 objective. 40

Figure 5.5. Task 5 objective 41

Figure 5.6. Task 6 objective. 41

Figure 5.7. Task 7 objective. 42

Figure 5.8. Task 8 objective. 42

1 Introduction

With the increase in production and the demand for software, more and more programmers are needed to meet this demand. This implies that more people will be in front of a computer causing these people to contract some type of injury in the upper limbs, because using the computer for more than 5 hours a day is enough to accumulate stress in certain parts of the body such as shoulders, elbows, wrists, hands and fingers due to repetition of movements. These diseases are known as *repetitive strain injury* (RSI), the recovery of this type of injury depends from person to person and from case to case, some taking from 3 to 6 months to really be recovered and in some cases leaves some gaps in the areas that were affected [1]. Another study [2] in a telecommunications company showed that 41% of workers, who were in front of a computer for more than 5 hours, felt fatigue in the upper back area around the neck and 38% in the shoulder area. Not only for people who have these kinds of "temporary" injuries, but also for people who have a permanent disability that prevents them from using the keyboard and mouse, which is necessary to write code.

This thesis is focused on providing a voice-based interaction to help people who face such difficulties. However, the combination of using the keyboard and mouse with voice commands is also something to explore, because there are tasks that might be easier to do with voice commands (e.g., such as navigation through the code or modifications to the visibility of members).

1.1 Goals

The main goal of this dissertation work is to make a functional prototype in which it is possible to provide voice commands for the Java language, whether for inserting statements, for navigating through the file, and for other basic commands such as creating class members etc.

After developing the proposed prototype, we aim at evaluating it with users, both with and without physical disabilities (if possible). These usability tests with people with physical disabilities have as a goal to understand if the prototype is intuitive to use, and if it can really replace the use of the keyboard and mouse. For users without disabilities, the objective is similar to the previous one, but to understand if it can be a tool that can be used for certain tasks because it is faster, for example, when navigating through the document and the project.

In the end, the goal is to compare the two methods, the use of the keyboard and mouse, the classic way, and the use of voice commands and understand in which aspects one is better than the other in terms of time to perform certain tasks and the effort needed to perform them.

1.2 Research Questions

- How can voice be used together with a structured editor ?
- How can voice be used for efficient code editing and how is it compared to the traditional way (keyboard and mouse)?

1.3 Contributions

A prototype editor called Jasay was developed, which consists of a structured editor that receives voice commands capable of altering Java language source code. The prototype consists of two parts, the first¹ with the speech recognizer, which aims to receive and transform commands spoken by the user into text.

The second² part consists of validating and executing the command if it is valid, for which it is validated taking into account two validation processes. The first is called the list of possibilities, which consists of a list containing words that could be in the command, thus validating the command, and the second is the use of a grammar using its rules to validate the commands.

The prototype was tested with users in order to receive feedback in terms of usability for first time users and aspects that could be improved taking into account their results.

Tests were also carried out comparing the prototype with the use of the keyboard and mouse to try to see if there is a big difference between the times and really where that time difference is.

1.4 Dissertation Outline

Chapter 2 presents related work based on other editors that use voice code modification mechanisms, speech recognizers, and projectional code editors. Chapter 3 presents how Jasay works in general without going into great technical detail. Chapter 4 explains Jasay in detail, going into implementation details. In Chapter 5, user tests are carried out. In Chapter 6, tests are carried out comparing the use of the keyboard and mouse with the prototype. Chapter 7 presents the conclusions and proposals for future work.

¹ <https://github.com/AlexCancelinha/Jasay>

² <https://github.com/AlexCancelinha/VoiceRecogClient>

2 Related Work

2.1 Code editors using voice commands

OpenSource system using normal language for programming

Désilets et. Al [3, 4] created an OpenSource system for voice programming that consists of the user using normal language following a certain syntax that is simple to learn. The following is an example of a voice command to type an if statement: "if current record number is less than max offset then" that creates (see Figure 2.1)

```
if (currRecNum < maxOffset)  
{
```

Figure 2.1. Example if statement created from the voice command "if current record number is less than max offset then".

The word "then" in this context causes the if condition to end and the cursor to be automatically placed inside the if block, where it is inside the if.

Another very interesting detail of this study was the fact that although it said "current record number" the system was able to understand that it corresponded to the short version with abbreviations, "currRecNum". Also on this part, it was able to find ways to correct errors in which the recognizer may not have been able to recognize the words that were said clearly.

Web system that uses voice recognition to write code

Gonze et. al [5] created a web system that allows the use of voice for writing code. For speech recognition several systems were tested from Google Cloud Speech, Amazon Transcribe, PocketSphinx, etc. In the end, PocketSphinx was chosen to be the system due to its ease of use, it was used together with a custom grammar. A grammar is a language definition that enables the recognizer to achieve a higher precision in recognition. The grammar defines the form of the commands beforehand and simply when a command is spoken the recognizer looks for the command that is more similar in the grammar. This is a point that needs improvement because what can be done with grammars is limited and is a powerful weapon when you know all the possibilities of paths that can be taken, which doesn't happen in the case of programming. Something to improve is the fact that this implementation divides a command into 2 phases to increase the precision of the commands, the first called the struct phase which refers to the use of specific words that will refer to the type of command that is being made and

the second phase, the "naming phase" which corresponds to the phase where the created instance will be renamed, whether it is a class, variable or function.

IDE named VocalIDE

Rosenblatt et. Al [6] developed an IDE that specializes in recognizing voice commands. First they conducted a study, which consisted of showing on one side a correct piece of code and on the other side the same piece of code with errors and the goal was to have the people correct the pieces with voice commands to try to collect what phrases people used to correct those errors. The study concluded that most of the words people used were in an attempt to position themselves correctly in the code, for example "jump to ...". With this feedback they built VocalIDE which allows users to dictate code either word by word or letter by letter, navigation can be done by lines of code, small mouse setting commands, and even by pointing out instance names.

VSCoDe extension that uses voice to code in JavaScript

Ribeiro et. Al [7] created an extension to be used in VSCode that allows JavaScript writing by voice commands. Speech recognition was based on the Azure system, which is a Microsoft product that makes the transcription from voice to text, a peculiarity of transcription is that the voice command is transcribed in blocks of 1 second instead of the typical let the sentence end. The command already in text receives a lot of treatment by removing stopwords and then matching the text command with one of the predefined commands that will then be introduced in the code. An interesting implementation is the fact that they can provide several possibilities to represent a command, for example to create "const a = 7" you can say "new const called a equals number 7" or else "const a equals number 7" which allows the learning curve to be smaller. But the system used compared to new systems, or even, one that existed at the time like Google's speech recognition system has a better performance than Azure's.

PyVoice IDE

Nizzad et. Al [8] implemented an IDE called PyVoice IDE that can understand voice commands such as "program to calculate simple interest" which is not a simple command but an advanced command that requires the system to recognize and know how to do this certain type of functions and for that in this study it is evaluated if the use of transformers is beneficial and as presented by the study they present very interesting results, but it is not revealed how we can navigate through the IDE nor if it will be possible what is essential in a type of system like these where the goal is not to use the mouse which is the tool used to navigate both the file and the project itself.

Multi-language capable of producing code using voice commands

Hossain et. Al [9] developed a system capable of producing code for three programming languages, these being Java, C and Python. For the speech recognition part, the Google speech API was used, after starting the program it is necessary to choose the language you want to use and then the system will be waiting for commands to be interpreted and consequently be added. An interesting specification is the fact that they can build and run the file also with voice commands, also able to execute console commands by voice. The authors also tested the time it took to do certain tasks, comparing the use of voice, the use of keyboard and mouse and both together, and the results showed the shortest execution time was both together, then the voice commands and lastly the use of keyboard with a big difference for the previous two. Something that was not specified was how the navigation in the code was performed, and that is a very important part because in this type of implementation there has to be a way to jump from one side to the other as is done in programming with mouse and keyboard. The conclusions of the work is that there are some speech recognition systems that are more efficient than the one that was used.

Java IDE that uses voice commands to change code

Shahane et. Al [10] created an IDE for the Java language that receives voice commands in which basically the user has a list of voice commands with examples of the types of sentences they have to say, which works as a guide, which when said are then processed using Natural Language Processing and Artificial Intelligence principles. One of the features of this implementation is that in the print case to write "System.out.println("This is test");" we only need to say "print this is test" which is a simplification in the print case that otherwise it would be quite painful to try to produce the command to dictate. Something that is not approached in this study is how to navigate through the file itself. Another thing that would really be beneficial is the fact that it would be possible to browse the project as well, something that is never talked about in the study.

HTML and CSS editor called Voiceeye

Paudyal et. Al [11] Voiceeye is an editor that combines eye, voice and mechanical switches in an attempt to write HTML and CSS. Voice commands are used to select, navigate and remove pieces of code as well as dictate some form of text that is not code, in this case comments. Gaze is used along with a keyboard that appears on the screen to type character by character, but as this form would be slow a complementary system Emmet was used which works as a plug-in that helps increase the speed of coding with code assistance. The mechanical switches were used to confirm and select either voice commands, or the on-screen keyboard inputs. Even with the assistance of the plugin in writing code, if voice was used instead of the gaze system and the mechanical switches it would be significantly

faster and more efficient, as noted by one of the testers in the usability tests that were done by the study. An interesting point is the use of voice as a tool for navigation in the code.

Java editor called VoiceJava that uses AST

Zan et. Al [12] developed a new programming language called VoiceJava that is prepared for commands to be dictated. In which a valid text command is received by the command pattern recognizer which creates a command pattern instance that will generate an Abstract syntax tree (AST) fragment that will be inserted into a Java AST using a direct syntax and then code is generated through that Java AST, which is a very interesting approach because it allows you to maintain a fairly good code structure but at the expense of user friendliness. This implementation was only tested for commands in text form and not with the use of voice. There are points to improve in this implementation, such as in case we are in the beginning of a java file where we are putting import statements we can't just create a class for example "define public class newClass" because we have to say "nextStep" first to go from an import insertion to a class insertion/declaration. Another point for improvement is the fact that you use more than one command, for example to create a variable, such as "int number" you need 2 commands one with "define variable number" and the other with "type int" when clearly you can do it all in one command such as "define int variable number".

2.2 Speech recognition systems

Comparison made between multiple speech recognition systems

Këpuska et. Al [13] compared commercial speech recognition systems such as Google speech API and Microsoft Speech API with CMU Sphinx-4 which is an open-source speech recognizer, using the WER-Word error rate as a comparison metric. The test consisted in sending audio files to the systems where 37% WER was achieved in the case of sphinx, 18% in the case of Microsoft Speech API and 9% in the case of Google, which was the best compared to the others for this test.

Automatic speech recognition named Whisper

Radford et. Al [14] created a speech recognition system called Whisper trained with 680000 hours of data with several languages from supervised data that was collected from the web, with this large amount of data they were able to increase the robustness in terms of accent, background noise and technical language allowing also transcription in several languages not being constrained to only one language. The architecture of whisper is simple, implemented as an encoder-decoder transformer. The incoming audio is divided into 30-second blocks that enter the encoder. Then the decoder is trained to predict the text that has been spoken that mixes special tokens that direct the model to identify

language, date and time in terms of within the sentence, and be able to tell if speech might contain more than two languages. The smallest model, in terms of parameters achieved a WER of 6.7%. The model achieves a 55.2% decrease in error when evaluated with other speech recognition datasets.

2.3 Projectional Editors

Projectional editors are editors that use the concept of projectional editing as opposed to traditional editors that use the concept of source editing. Source code editors consist of modifying the source code, which in this case is raw text, to be source code editors they need to facilitate the modification of the same by providing features such as syntax highlighting, indentation, autocomplete and some others. Source code combines the editable and storage representations and storage representations, editable representations represent what you can edit in order to change the system and the storage representation the persistent record of the system definition. When it is necessary to execute the source code it is transformed into its executable representation, to achieve this step, usually the source code is transformed into an abstract representation during the compilation phase, making the source be the core definition of the system.

In the case of projectional editing the abstract representation is the core definition of the system, where the editor has tools that alter this abstract representation. The editable representation and the storage representation are completely separate and independent, which is the major difference between projectional editing and source editing.

Prong projectional JSON editor

McNutt et. Al [15] proposing a lightweight projectional JSON editor prototype named Prong, in order to create a structure editor for JSON it was necessary to develop a mapping from a JSON Schema to a graphical structure editor and then to create an API that allows application designers to concisely define custom views.

Javardise

Santos created a prototype of a structured editor named Javardise [16] that supports part of the Java language syntax. The idea of this editor is to make it so that the user almost never sees code that is syntactically wrong, to make it easier to view the code by automatically indenting it, and to automatically add semicolons. The prototype is based on a model-view-controller architecture, in which the editor basically reacts to changes in the model by changing the view, in this case the editor view, according to changes in the model.

The editor uses the ASTs of a library called JavaParser as the model of the editor. JavaParser is an open-source library written in Java that offers APIs for analyzing, manipulating and generating Java source code. One of the main use cases is code analysis, where JavaParser analyzes source code and builds an abstract syntax tree (AST) that represents the structure of the code. This AST can be traversed to extract valuable information such as identifying class hierarchies, method invocations or variable declarations. With JavaParser, developers can analyze and extract information from code, generate code automatically and perform refactorings.

It is also useful for static analysis and bug detection, helping developers to improve code quality and development efficiency. Another great advantage of JavaParser is its code generation capabilities. Developers can create Java source code programmatically by building the AST using JavaParser's APIs. This functionality is especially useful for generating repetitive or routine code, such as getters and setters, builders or even entire classes. Using this library developers save time and reduce the risk of errors when generating code.

In order to be able to see the changes to the model made by using Jasy, Javardise is used as a visualizer to see these changes.

3 Approach

Our goal is to develop a prototype of a structure editor that is able to receive voice commands in order to be able to navigate and modify the code. This prototype is a system composed of two fundamental parts. The first part consists of a speech recognition component to convert spoken language into text. The second part of the system involves the processing and interpretation of the results obtained in the first stage, in order to output valid code statements (see Figure 3.1).

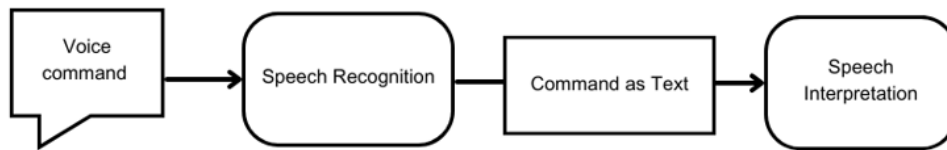


Figure 3.1. General overview of the system's main components.

3.1 Speech Recognition

The aim of the recognition model is to be able to convert the commands that the user is saying into sound waves and convert them into text to be processed later. A component like this is important because it allows us to transcribe speech into voice quickly and with much less transcription error. For a system of this type, in which timely interaction is a very important component for the usability of this type of system, transcription must be quick and assertive, as an error in transcription causes the user to have to repeat the command, generating stress and wasting time.

Our work will rely on an external speech recognizer, on which we have no influence in the implementation, as we only define configuration settings and selected which models to use. Models vary in size, and there is a tradeoff with respect to the need for the recognizer to be fast and the transcription error to be smaller.

To this end, we investigated 3 speech recognizers, Google Speech API, CMUSphinx and Whisper, to evaluate their transcription speed and their assertiveness in relation to the type of commands we are expecting. After some experimentation, it was decided to go ahead with the Whisper recognizer. This choice was made because Whisper has a less word error rate when transcribing, another advantage it has is that it was trained with technical language making less errors when transcribing compared with another's speech recognition systems.

Whisper was produced by OpenAI [14] trained with 680000 hours of supervised multitasking data collected from the web. Due to its large and diverse dataset the model is quite robust to accents, background noise and technical language.

There are several types of models that vary in size and what they can do, from the "tiny" model, with 39 million parameters, to the "large" with 1550 million. As is obvious the use of larger models is possible to acquire a higher hit rate in the words that are being said sacrificing processing time, for the prototype the "small" model was used which consists of 244 million parameters.

The user's voice commands are captured in time intervals from the moment speech is detected until about one second after no speech is detected. Calibrations are made to the speech collector taking into account the space where the user is, this calibration being done when an audio with speech is collected by the user.

Once the audio is captured, the recognizer is in charge of transcribing the audio containing speech to text, thus completing the first part from the speech capture corresponding to one by the user to the transcribed text that is sent to the second phase.

3.2 Speech interpretation

The information received from the speech recognizer has to be filtered in order to confirm if what the user is saying makes sense taking into account what we are programming. For this a list of possibilities of tokens that are accepted was used, giving an example, if in the list of possibilities we have the word "int" and if the user says "int x equals 0" as the word "int" is in the list of possibilities the voice command passes the filtering phase of the list.

The other method of validating commands is to use a grammar to detect other types of cases that the list of possibilities can't handle.

If the command is valid it will be executed by the interpreter making the changes that correspond to the command said by the user, in the other part if the command is not valid it will be ignored (see Figure 3.2).

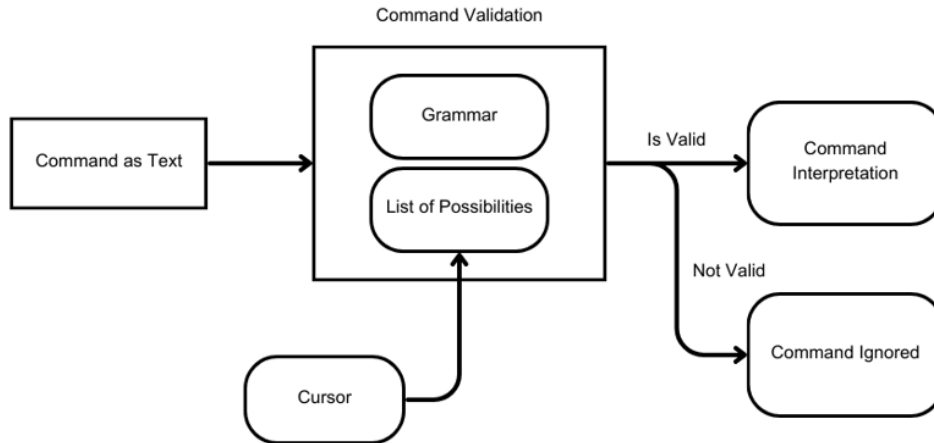


Figure 3.2. General overview of the procedures in the Speech Interpretation part.

3.2.1 Command Validation

This step is done when it is received from the speech recognition part, which has the function of knowing if the command is valid, and it is necessary to know if what was said makes sense to be in the cursor location. To do this, two strategies were created: filtering by a grammar, and by using the list of possibilities.

3.2.1.1 Command grammar

When dealing with situations where you need to validate commands that the user has more freedom, our choice was to use grammars.

A grammar defines the syntax and structure of a programming language or data format, providing rules for writing valid code or data, allowing language processing tools to analyze and manipulate that information effectively. In this case, it is used as mentioned above to create rules so that we do not require the user to follow a predefined recipe, thus giving the user the freedom to say commands in a natural way.

We used grammar for some types of operations that could be done with the list of possibilities as well, but we would have to use as mentioned above a kind of recipe that would have to be taught to the user, worsening the usability of the same. The grammar used is shown below in Extended Backus-Naur Form (see Figure 3.3).

```

grammar CommandsGrammar;
classCommand:
(Text ' '* ClassCommandKeyword (' PrivacyKeyword)? (' StaticKeyword)?
(' FinalKeyword)? ' TypeKeyword name;
classCommandSetterAndGetter:
(Text ' '* TypeToCreate name;
addParameterInMethod:
(Text ' '* (Parameter ' ') TypeKeyword name;
navigationCommandSimple:
(Text ' '* NavigationCommandKeyword (' Numbers)? (' Text)*;
name:
(' Text)+;
Parameter: 'parameter' | 'argument';
TypeToCreate: 'setter' | 'getter' | 'gather' | 'constructor';
NavigationCommandKeyword: 'up' | 'down' | 'left' | 'right' | 'enter';
ClassCommandKeyword: 'method' | 'field declaration';
PrivacyKeyword: 'private' | 'public';
StaticKeyword: 'static';
FinalKeyword: 'final';
TypeKeyword: 'int' | 'float' | 'double' | 'boolean' | 'long' | 'short' | 'char' |
'string';
Text: [a-zA-Z]+;
Numbers: [0-9]+;

```

Figure 3.3. Grammar used in the prototype.

The operations in which the grammar was used were for when we have the cursor in the body of the class and we intend to create a member of the class in this case it can be a variable or a class method with this purpose the “classCommand” was created to allow this situation.

In order to have a shortcut for getter, setter or constructor methods, the rule “classCommandSetterAndGetter” was created.

It is possible to realize the creation of the getter and setter methods of class attributes from the rule “classCommand” as well, but this rule is a shortcut as it allows the user to create the above methods with much less work.

It's also possible to add parameters in the methods using the rule “addParameterInMethod” of the grammar making it possible for the user to use an extended vocabulary to add parameters to a method.

Finally, the rule “navigationCommandSimple” was used to deal with navigation, i.e. the way we move within the editor.

3.2.1.2 List of possibilities on cursor position

The list of possibilities was one of the ways used to address the problem of how to ensure that what is being captured as speech actually contains something useful to be processed for the code given the cursor position.

The list of possibilities to act as mentioned above needs to be always changing taking into account the context in which it is inserted, that is, it is necessary to take into account where the cursor is located. The content of the list of possibilities is obtained by syntactic analysis, i.e. the content of the list only contains the keywords that are valid for the field the cursor is in. The list of possibilities is dynamic i.e. every time the cursor changes location the same list is changed, because when we have the cursor on a type identifier the list of possibilities will be different if instead we are in a field where what is expected to appear is an expression.

The list of possibilities as mentioned before changes according to the cursor, that is, when we have the cursor on an access modifier, in this case, "public", "private" or “protected”, what is in the list of possibilities are only the words reserved for these modifiers, because it does not make sense to say anything else besides these modifiers (see Figure 3.4).



Figure 3.4. Example of when the cursor is in an access modifier.

The same happens when we have the cursor over a type that can represent both the type of a variable and the type that is expected at the output of a method. Again, only the words reserved for primitive types and the non-primitive type “String” are inserted in the list of possibilities, as well as types that

are contained in the project, that is, for example, we have a class named "Player" this type will also be present in the list of possibilities (see Figure 3.5).



Figure 3.5. Example of when the cursor is in a type of a variable.

It is not necessary to place arrays in the list of possibilities because, in order to create an array, it must be of a valid type and the list of possibilities will automatically validate it.

In case the cursor is on the name of either a method or a variable declaration, the list of possibilities will be empty because in this case as the user is expected to name the declaration it is not possible for us to predict what name the user wants to give ensuring that this freedom is not lost (see Figure 3.6).



Figure 3.6. Example of when the cursor is in a name of method declaration.

When we are with the cursor inside a method on a blank line we call these cases Empty Statement, in this case what we can put inside methods in Java are typically referred to as method elements, in this case it is allowed to add variable assignment statements, control flow statements, function calls, exception handling. This is introduced in the list of possibilities, the reserved words that represent the primitive types, along with the names of classes that can be instantiated that are contained in the Java files of the project (see Figure 3.7).

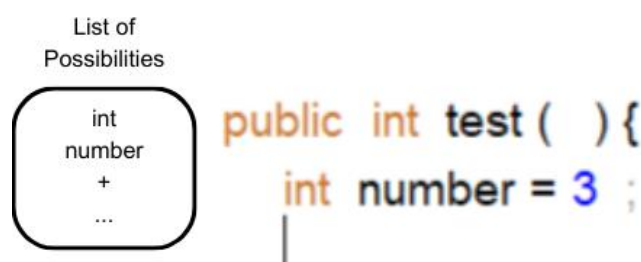


Figure 3.7. Example of when the cursor is in a statement, in this case an empty statement.

To deal with the variable assignment part, in the list are also placed the reserved words for flow control in order to be possible to use control flow statements, to deal with the possibility of having function calls are also introduced the functions of the file that is being edited together with all the functions of the project that have the access modifier "static".

The list of possibilities when we have the cursor over an expression that can be either a variable assignment statement, or an expression inside a control statement such as the "if" condition for this the list of possibilities allows a variety of possibilities that are not delimited by the type that is expected in that expression.

Giving again the example of an expression of a condition of the control statement "if" that needs, at the end of calculating the expression, being of boolean type does not mean that what is used inside the expression is of that type, for example, having "1 == 2" is an expression of boolean type but in fact it is a comparison of two integers.

To be able to guarantee that it is possible to make this type of comparisons the list of possibilities consists of all global variables, local variables, arithmetic, comparison and logic operators and together with function names that have some type of return, thus guaranteeing freedom (see Figure 3.8).

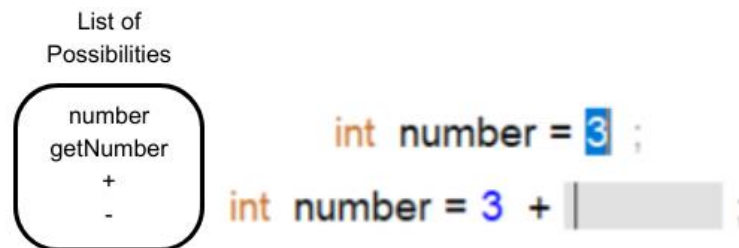


Figure 3.8. Example of when the cursor is on an expression. The first image shows an expression composed only of a number and the second an expression composed of an arithmetic expression.

3.2.1.3 Filtering procedure

When a command is received, it is separated into words that are compared with the words in the list of possibilities. This comparison between words is made using two methods, the first being the distance between words, "Levenshtein distance", and the phonetic distance between words.

When a word in a command is validated, it is replaced by the word that matched it in the list of possibilities. When a word is validated, it automatically validates the command, but validation continues for the other words in the command. This is because it allows words that may have been

perceived incorrectly by the recognizer or unintentionally by the user to be replaced, also acting as a corrector.

Levenshtein distance

The Levenshtein distance calculates the smallest number of single character edits in this case insertion, removal or substitution, these criteria having equal weights in this case 1 each, i.e. in a concrete case the Levenshtein distance between the string "hello" and "hallo" is 1.

The command received is divided into words and validated one by one. This validation is done by comparing the separate words in the command with the words in the list of possibilities. The word when compared is valid if it has a levenshtein distance of less than 1 and consequently the command is validated if one of the words is validated. When we're talking about words that are less than or equal to 2 characters long, the distance must be 0 to be validated.

When the word is validated, it is replaced by the word from the list of possibilities that matched it.

Giving an example, with a list of possibilities, ["int", "for", "while"] and if the command spoken by the user is "whale", either by a transcription error or by mispronunciation, the command is split into words, in this case only one, and is validated because the distance between the word "whale" and "while" is 1 and the word "whale" is replaced by "while" in the command spoken by the user.

Phonetic Similarity

Phonetic similarity was also used to validate the command with the list of possibilities, but unlike Levenshtein distance, which is the quantitative difference in the minimum number of characters modified between two words, what phonetic similarity does is compare words according to their pronunciation or phonetic representation being a very good way to compare words that are spelled differently but have the same pronunciation or phonetic representation.

The comparison is again made word by word. Both words are transformed into their phonetic form and compared, giving an example, the word "while" is transformed into "W AY L" which corresponds to its phonetic form, if the forms are the same this means that the words have the same form being validated the word is replaced by the word that matched in the list of possibilities.

3.2.2 Command interpretation

After the command has been validated either by the list of possibilities, or by some of the rules of the grammar or other types of command validation, we realize where the cursor is located in order to

direct the command to the correct type of processing depending on the location of the cursor, if it is inside the body of a method of the class or if it is inside the class but outside a method, and depending on the type of command, it can be navigation or modification of the Java file.

It is possible to make a certain group of commands by voice in the editor, but they follow some rules, for code insertion the user is supposed to say the code, it is not necessary to say complete code, that is if we want to insert at the cursor location it is not necessary to say parentheses, commas or semicolons something like "if open parentheses..." is simplified with just saying "if" and automatically the editor will add the parentheses.

There are some rules that the user has to follow such as following a pattern as if he were dictating code, so commands such as "insert an if..." are not possible to be entered in the editor, although they are validated by some of the command validation processes.

In the case of commands that are for navigation through the editor, as long as the grammar condition is met the command is accepted and executed, so it is allowed in these cases that there is a certain freedom on the part of the user to be able to navigate the editor in a flexible way, the same happens for the other rules of the grammar, such as changes in the visibility of methods and attributes for example, because they are different requests making it not necessary an increased learning on the part of the user.

3.2.2.1 Commands for changing the class

You can modify the class by placing the cursor in the body of the class, and you can create field declarations and methods, including setters, getters and constructors in the case of methods. Some of the possible commands are shown in Table 3.1.

Table 3.1. Commands allowed within the class body.

Location	Type	Variations	Example	Final result
Inside the class body	Field Declaration creation	Standard	"create a field declaration int number"	public int number;
Inside the class body	Field Declaration creation	Standard + Visibility	"create a field declaration private int number"	private int number;
Inside the class body	Field Declaration creation	Standard + final	"create a field declaration public final int number"	public final int number;
Inside the class body	Constructor creation	Standard	"create constructor student"	public Student(){}

Inside the class body	Getter creation	Standard	"create getter number"	public int getNumber(){return this.number;}
Inside the class body	Setter creation	Standard	"create setter number"	public void setNumber(int number){this.number=number;}
Inside the class body	Method creation	Standard	"create method int add to school"	public int addToSchool(){}
Inside the class body	Method creation	Standard + Visibility	"create method private int add to school"	private int addToSchool(){}
Inside the class body	Method creation	Standard + final	"create method final int add to school"	public final int addToSchool(){}

3.2.2.2 Commands for changing the method

When we are in the body of a method, we can use commands to create statements, expressions and add arguments to the method.

It is possible to create expression statements and control-flow statements. Table 3.2 shows the possible commands for manipulating statements.

Table 3.2. Commands to modify statements allowed within the body of the method. STC corresponds to the space that the user can change.

Location	Type	Variations	Example	Final result
Inside the method body	Statement Creation	Standard	"int name equals 3"	int name = 3;
Inside the method body	Statement Creation	Without expression	"int name"	int name = STC;
Inside the method body	Statement Creation	Without expression and name	"int"	int STC= STC;
Inside the method body	Control-Flow Statement Creation	Standard	"if"	if(STC){}
Inside the method body	Control-Flow Statement Creation	Standard + expression	"if name equals 3"	if(name == 3){}
Inside the method body	Return Statement	Standard	"return"	return STC;
Inside the method body	Return Statement	Standard + expression	"return result"	return result;

When the cursor is located in an expression, the user is given different ways of manipulating the expression as shown in the Table 3.3 :

Table 3.3. Commands to modify expressions. STC corresponds to the space that the user can change. CV corresponds to the value where the cursor is placed.

Location	Type	Variations	Example	Final result
Inside expression	Expression Creation	Standard number Call	"eleven"	11
Inside expression	Expression Creation	Standard variable Call	"name"	name
Inside expression	Expression Creation	Standard Function Call	"get number"	getNumber()
Inside expression	Expression Creation	Standard Operator Call	"seven plus number"	7 + number
Inside expression	Expression Creation	Operator Call with one value	"seven plus"	7 + STC
Inside expression	Expression Creation	Operator Call no values	"plus"	STC + STC
Inside expression	Expression Creation	Operator call no values if the cursor is a value	"plus"	CV + STC
Inside expression	Expression Creation	Operator call with one value if the cursor is a value	"plus seven"	CV + 7

4 Jasay prototype

The prototype consists of a structured editor capable of modifying Java source code using voice commands, named Jasay. The prototype has been implemented in two separate processes that communicate through sockets with TCP connections. Figure 4.1 presents an overview of Jasay's architecture.

This was made because the speech recognition model is developed in Python and the library, that transforms the Java source code into the AST structure, is made in Java, making the command interpretation part being developed in Kotlin. The first attempt at implementation was using just one Kotlin process, using libraries that allowed Python processes to be created within Kotlin itself, but the time taken by the library with the process compromised the usability of the prototype. We then tested the use of sockets to transport information between two processes, one in python and the other in Kotlin, which took much less time compared to using the previous library.

One process is in charge of the speech recognizer model, which was implemented in Python, with the aim of receiving the voice command from the user and processing the voice by transcribing it into text.

The other process handles the user's command in text format and interprets it into a command that modifies the editor code. The position of the cursor in the editor and the current AST determine whether the command can be carried out. After the validation, the command is executed by modifying the AST.

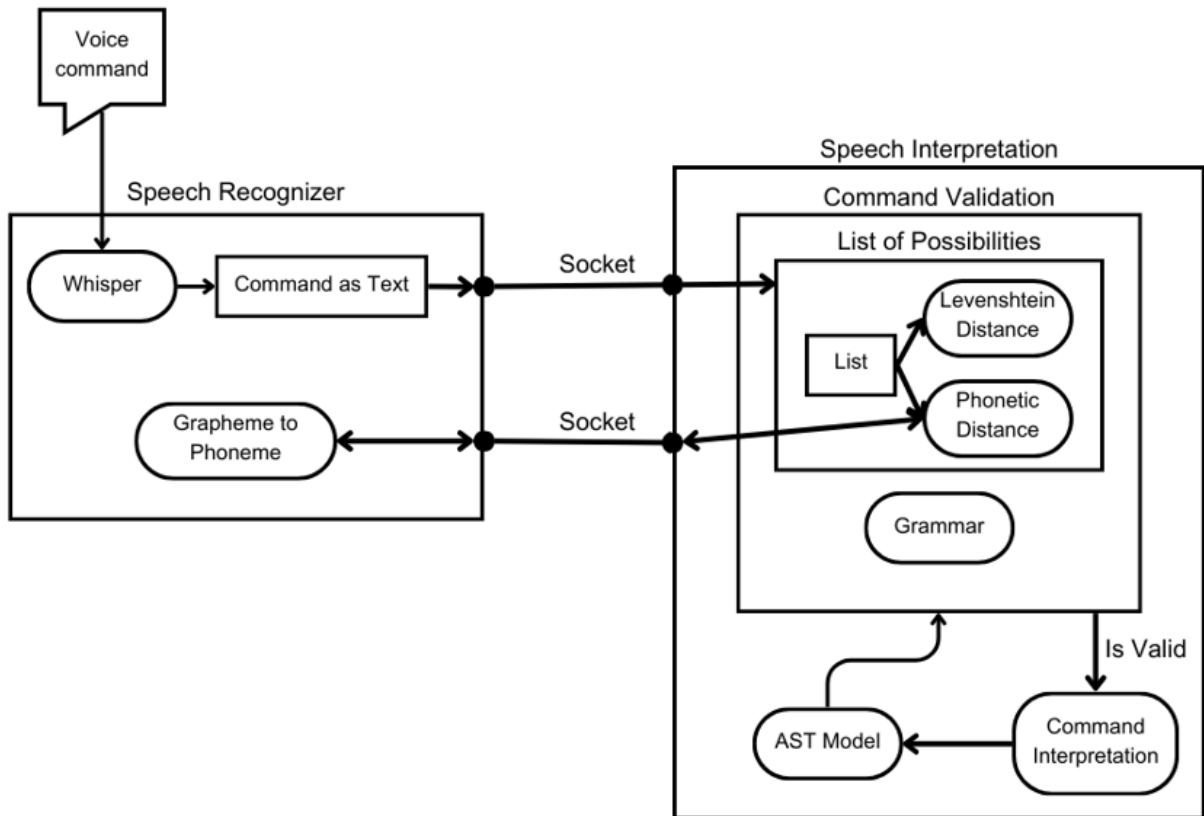


Figure 4.1. General outline of Jasay implementation.

4.1 Speech Recognition

The speech recognition part consists of using the speech recognition model in which the so-called "small" model was chosen, that consists of a model with 244 million parameters. When compared to the smaller data model of only 39 million is a significant increase in terms of parameters.

Using this model with a larger parameter dimension required using the GPU so that it is possible for the model to be fast enough so that the usability of the prototype is not compromised.

In concrete terms, in the vast majority of cases, the size of phrases spoken by the user will be small such as "int i equals seven". Using the CPU on average takes about 3 seconds to transcribe from sound to text types of phrases with this dimension or a little larger. For the GPU it takes about 0.3 seconds, which makes it possible to choose a model with greater parameters than was originally intended to be used. Being able to use a model with a lower word error rate, minimizing transcription errors by being more sure of what the user is saying, and consequently reducing error correction by the application itself in the command analysis part, reducing frustration on the part of the user.

To capture what the user is saying, a library is used to capture audio as soon as it detects speech from the user and finishes collecting when there is no speech for 0.9 seconds. For this collection we take into account the environment where it is, giving the possibility to calibrate the speech capture for more or less noisy environments.

This calibration is possible by changing the "energy_threshold" field. A high value of this field allows the recognition to be less sensitive which is useful for noisy places. In addition to this "setting" that the user can change, when the library finishes an audio collection a calibration is made to the sensitivity of the library so that it is always ready for changes in the noise level caused by the change of environment.

When audio is finally collected by the library it is sent to the specific speech recognizer, Whisper, where we receive a transcription response, containing various information and metrics, including the proposed transcription of what was said to text and the probability that nothing was said (no_speech_prob). This is because it is possible that the audio collection library picks up noise that does not contain speech. The audio collection is sent to the recognizer and the proposed speech-to-text transcription in those cases is noise, containing nonsense words and sometimes with strange symbols. With the help of this field, you can filter out the small noises that the microphone picks up.

When this filtering step is passed we are then sure that something was really said by the user, whether this is directed to the editor itself, or simply having said something. The next step is to pass this information from the speech recognition, responsible for all of the above, to the process responsible for the speech interpretation, this phase of passing the information of what the user said between the different processes is done via sockets which are connections made by TCP.

4.2 Speech Interpretation

The Java process has the role of dealing with the command transcribed by the recognizer in the speech recognition process, trying to figure out if the command takes into account the location of the cursor. The command is received via socket, then the command is filtered and if it passes this filtering the command is interpreted by the process by changing the Java code. The JavaParser library was used for the filtering and code modification.

4.2.1 JavaParser

The Javaparser is used to find out in which type of node the cursor is located, for example, when we have an expression "int i = 2;" and the cursor is located in the "int" we are facing a primitive Type. With

this information, the list of possibilities will be changed, this list consists of a first filtering of the commands said in this case if the command said by the user is not contained in the list of possibilities it means that it cannot be used. The library is also used to enter the commands from the AST which facilitates the introduction of commands because it is done directly because instead of dealing character streams in text files, we are only changing the tree.

4.2.2 Javardise

Jasay presents the Java code in the editor using Javardise's widgets, which react to changes in ASTs. Hence, the command actions that modify the code produced by Jasay are expressed as AST transformations, making it easy in the aspect that the interconnection between the editor and the so-called "writing" of code by voice because the editor is basically a view of the AST in memory corresponding to the Java code that is being edited. That is, when a change occurs in the in-memory AST, the view will be changed according to that modification seamlessly.

4.2.3 Command grammar

The use of grammars was made possible by using the Antlr4³ library, which generated the grammar parser automatically. To do this, it was necessary to create the grammar following the notation of the library itself. The rules were created according to the rules for creating grammars and then saved in a file with the extension "g4", where the library then creates the Java files with the parsers for the rules (see Figure 3.3).

4.2.4 List of Possibilities

As mentioned earlier, the list is used to filter what the user says, so that if the command spoken by the user contains the keywords that are in this list, the command is validated and it can be executed successfully.

The list is filled in taking into account the location of the cursor in relation to the model's AST. To do this, it was necessary to go through the AST to remove information such as variables, functions and classes in order to fill in the list taking into account the given position of the cursor (see Figure 4.2).

³ <https://github.com/antlr/antlr4>

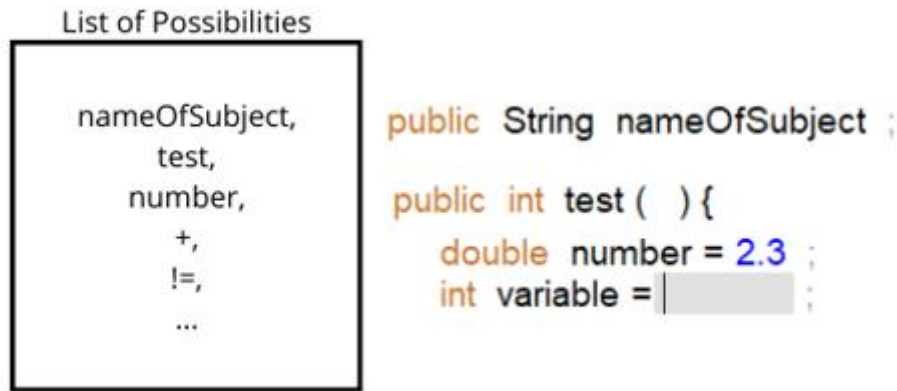


Figure 4.2. Example of a list of possibilities getting information from the AST.

4.2.5 Filtering procedure

4.2.5.1 Levenshtein distance

The command received by the speech interpretation is divided into strings with just one word and validated one by one with the ones that are in the list of possibilities. Figure 4.3 shows an example of using the Levenshtein distance between two words.

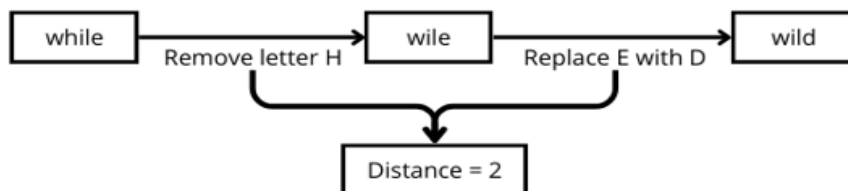


Figure 4.3. Levenshtein distance between the word “while” and “wild”.

As mentioned above, for strings with 3 or more characters, there is a threshold of 1 at which the word is considered valid. There is no threshold for words with less than 3 characters because if the threshold were used, words with this number of characters would be very similar, which would lead to many validation errors.

A dynamic threshold strategy was also tested, which consists of having a threshold but taking into account the length of the string. In this case a word like "collision" would have a higher threshold than a word like "if" but it ended up not being used because if the word was large enough the threshold

would validate the word with a completely different one just because the threshold would be big enough for that to happen.

4.2.5.2 Phonetic Similarity

This step was done thanks to a library [17] that deals with the transformation of grapheme to phoneme in Python (see Figure 4.4), so again as in the case of the recognizer the communication between the speech interpretation process and this new process is carried out by sockets.

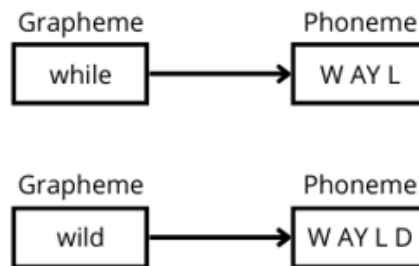


Figure 4.4. Words “while” and “wild” in their form of grapheme and phoneme.

That is, when there is a comparison between a word and the list of possibilities both are sent to the Python process that will perform the transformation to phoneme and send it back to the Java process.

After receiving the results with the phonemes the list of possibilities is saved together with the corresponding list of possibilities in phoneme form so that it is faster processing. This transcription process even if fast is more efficient if it is done in this way because as the project grows the list of possibilities will contain more variable names making that on a large scale there could be the problem of increasing the processing time that becomes waiting time for the user, something that in this case for an editor of this type it is convenient that it is very responsive.

Once the word and the list of possibilities are received they are both compared and if the phonetic composition of the words are identical the word is validated.

4.2.5.3 Camel case

The list of possibilities contains words that can be function names and variable names, and the convention for names in the Java language is camel case. That means that variable names like "student number" in camel case would be "studentNumber", which means that when the speech recogniser receives from the user "student number ...", the recogniser does not have the ability to realise that in this case it is necessary to concatenate two or more words to form a function or variable name. With

this in mind when we are filtering using the list of possibilities it is taken into account that a word of a command can be the beginning of another that is in the list of possibilities written with the convention.

Taking the example that was given above, when we have the command "student number ..." and the list of possibilities contains "[studentNumber]", the first iteration is made looking for the word "student" in the list, but as it does not contain the word is not validated but it is checked if the word that was not accepted, is contained at the beginning of any of the words in the list of possibilities. In the case that it is not validated and there is a word that begins with the word it is provisionally concatenated the word that we were comparing with the next of the command. With this in mind the two words in the command will be replaced by the concatenated one.

In the case of filtering with phonetic distance, it is not possible to treat these cases because they are combinations of alphabetic characters that are not technically the typical words that are used in everyday life. Therefore, there is no record of the sound and phoneme analysis that make up the word, so phonetic similarity cannot be applied to this specific problem.

It was explained above that the filtering begins with the word-by-word separation of the command said by the user, but there is another particularity that is when we are talking about some operators, such as "<" is written as "lower than" being made up of more than one word. In these cases, a validation of the list of possibilities was also constituted for the possibility of the entire command being validated instead of word by word to be able to make the case mentioned above possible. The process of validation is the same, but instead of a word, all the command is trying to match something in the list.

4.2.5.4 Numbers

Another problem that was found during the creation of the editor was how the numbers should be validated because the numbers were not contained neither in the grammar nor in the list of possibilities, which makes it not possible to use numbers in the editor. To counteract this problem a command is validated if it contains numbers, although the solution seems much less restricted than the other two that are used after this filtering process, there are other types of filtering that protect in concrete if the command that the user is wanting to use is valid in the position where the cursor is placed.

The treatment of the numbers had to take into account the fact that the user may want to use numbers in written format, that is, inside Strings. One of the possibilities that could have been implemented was depending on where the cursor was going to be interpreted if it would be placed as a written number or as a number in its numerical form. This implementation is quite restrictive, not

giving the user the possibility of being able to place a number in string format to know the length of the string thus obtaining in a different way to know how many digits the number has.

What was chosen to be the best resolution to the problem is to have a Boolean variable that decides whether the number should be written in numeric form or in string form. It is possible to change the value of this variable with a voice command by the user to change between the two types of modes.

One of them transforms all the numbers that are written in their long string form to numbers. This transformation is done by filtering, within the command, the number and then creating a string corresponding to the number taking into account the number of digits.

The second way transforms numbers from numeric form to their long form, in string. This process is done again as above by filtering the command leaving only a certain type of words that correspond to the number written in full.

After that it is processed and put into numbers by analyzing the string. With this form it is possible to transform both integer numbers, but also decimal numbers and also some variations of writing the numbers themselves such as "twenty five" and "twenty-five" are interpreted in the same way having the same result of the transformation as 25, this part was made using the library⁴. After the transformation is done, the command passes to the command interpretation processes to be executed.

4.2.6 Navigating the Editor

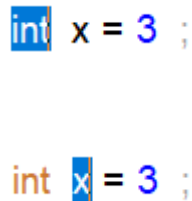
Navigation in a system where there is no possibility for the mouse and keyboard to place the cursor where we want it. Not having this possibility and navigation being an important part of it, it has to be done efficiently and in a way that does not cause fatigue in the user.

The navigation as said in the above parts of the filtering is detected with one of the rules of the grammar that allows the user to do the tasks performed by the arrow keys, "up" to move the cursor up, "down" to move it down, "left" to move the cursor left and "right" to move it right.

One problem that has been solved by using the Javardise code editor is the fact that when we have the cursor in a certain place when we execute a navigation command we do not walk with the cursor one character at a time but a whole block. The following example, of being in a line "int x = 3;" where the cursor is in the word "int", when we execute the command to move the cursor to the right,

⁴ <https://github.com/jgraham0325/words-to-numbers>

the cursor will not move to the next character after the int, which in this case would be the space between the "int" and the "x", but to the next area that can be edited by the user allowed by the editor in which in this case will be the "x" (see Figure 4.5).



The figure consists of two vertically stacked screenshots of a code editor. The top screenshot shows the code `int x = 3 ;` with a blue cursor block positioned over the word `int`. The bottom screenshot shows the same code with the blue cursor block positioned over the variable `x`.

Figure 4.5. In the first image the initial state of the cursor is in the statement type, in this case "int", in the second image after executing the command to move the cursor to the right the cursor moves to the name of the variable, "x".

With this navigation mechanism the editor allows it not to be too tiring for the user because moving the cursor character by character would mean going through the various stages from the speech recognition part to the speech interpretation part. Although these steps are quite fast to process, there is always some processing time and if there was no such possibility to navigate through the code in a fast way the usability of this editor would not be as easy and fast to use as a usual code editor.

Another navigation mechanism that was used in this editor is the fact that we can use a command from those mentioned above but several times. We can repeat a command a certain number of times, which makes navigation much faster, because in any instance we can see where the cursor is and how many times we will have to repeat a command.

Another aspect that had to be taken into account when dealing with this part of navigation is the fact that there is not only a certain phrase to trigger this type of navigation mechanism but the user can say it in several ways thus reducing the learning load needed at the beginning when it is necessary to learn how to interact with the editor.

4.2.7 Status Feedback

Feedback has been implemented in the code editor that aims to give the user information regarding when the speech recognizer is ready to pick up the voice command and when it is processing the voice command to text(see Figure 4.6).

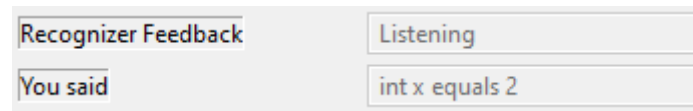


Figure 4.6. Status feedback showing the recognizer listening and the last command said by the user.

This is important because if the user doesn't know when the speech recognizer is ready to receive the audio it can result in synchronization problems between the recognizer and the user(see Figure 4.7).

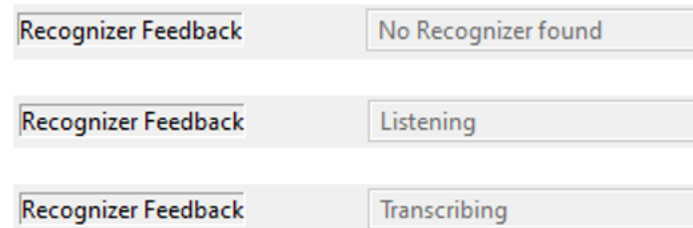


Figure 4.7. Different status of the recognizer.

A space to show what was transcribed by the speech recognizer was also implemented giving the user the proposal of what was said. This makes sure that the user can have a sense of what was perceived by the recognizer and can check if he really understood well or not.

Both of these status feedback forms help the user not to lose patience when using the application, one helping with the synchronization and basically saying when is ready to receive commands, and the other to get some feedback on what was transcribed.

4.2.8 Commands for changing the class

When the cursor is located in the body of the class and the command is validated, it is ready to be processed. The interpretation of the commands depends on the location of the cursor, in the case of being in the body of the class there are two possibilities, the possibility of the command having been validated by one of the rules of the grammar or being validated by the list of possibilities. When it is validated by one of the rules of the grammar, the user is trying to insert a field declaration or a method, the latter can be a getter, setter or constructor.

As a grammar is used, the information of the command is expressed in the grammar rule itself, for example, the command said by the user was "create method public static int example" which will be validated by the "classCommand" rule of the grammar. The rule extracts the type of property to create, method or field declaration, the access modifiers, the non-access modifiers, such as "static" or "final",

the type and the name. The name has a particularity that was mentioned above that allows, even with more than one word, to be detected and placed according to the camelcase convention.

With this information, a method is created with the information and then inserted in the in-memory AST. In the case of the field declaration the process is the same, since the detection rule for methods is the same because both follow the same structure when the field declaration does not contain the initialization part and when the method does not contain parameters.

Figure 4.8 shows that both have the same structure so it was necessary the user input to be able to distinguish which of the two the user wants to introduce. If the user's intention is to create a method the user should say something like "create me a method public static int factorial" and in the case of the field declaration something like "create me a field declaration public static int factorial".

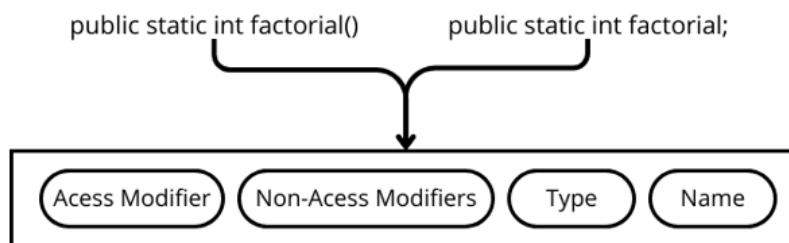


Figure 4.8. The structure of the field declaration with no equals and for a method.

When the goal is to create a method of type getter, setter or constructor, is again a command in which it is validated with a rule of the grammar. The command contains information about the type of what structure is to create, the modifiers, either non-access modifiers or access modifiers, and the name, in the case of the setter and getter.

In the case of getter and setter we also receive the name of the field declaration that the user wants to do the getter or setter, with the information collected the type of method requested in the command is added to the AST.

This is not the only way to create methods of the getter and setter types because they are normal methods like the ones that are talked about above. This is just a tool that is provided to the user because it is something that is quite common in today's IDEs, thus managing to automate this type of situations that in the case of when you are providing voice commands, the amount of commands needed is reduced and as a consequence the speed which you can "write" code increases.

Also in the context when the cursor is in the body of a class it is possible to create a field declaration with initialization, this is something like, "private int number = 7". This is not possible to do with the rule of grammar mentioned above because it is possible to have a field declaration that has

an assignment, providing an initial value to the declaration that is validated by the list of possibilities. After the command is validated, it is checked for the existence of the word "equals"("=") because this is the verification that it is a field declaration with assignment. Then information is collected by manipulating the string, in the case of visibility modifiers and for Non-Access modifiers, the first part of the string is analyzed, up to the character, "=", looking for the reserved words of these types of modifiers. After removing the modifiers, the name is everything that is left of the string until the "equals" ("="), in the case of the attribution part it is everything that is after the equal ("=") as the assignment part is an expression, its treatment is explained below in the Expressions section.

4.2.9 Commands for changing the method

The validation of the commands that change methods are almost all done according to the list of possibilities, checking if any of the words are contained in the list or if it contains numbers.

Commands that change the parameters of the method are not validated by the list of possibilities, as they are validated by the grammar rule "addParameterInMethod". In this case, when the command is validated, we can extract the type and name of the parameter we want to create. With this information we change the AST by adding the parameter to the method in which the cursor is located.

After being validated and the cursor is inside a method, the command is redirected to be processed, for this the processing is divided into two parts: statements and expressions.

Expressions can also be present in a field declaration that is initialized. These field declarations are detected because they contain the word "equals"("="), where everything to the right of this identifier is as an expression and is processed in the same way as those in statements.

4.2.9.1 Statements

When we have the cursor on an empty line, which in reality Javardise treats those as an empty statement (";"), it is possible to create statements in different ways.

The creation of assignment statements that consist of assigning value to variables, in the case of "int variable = 2;", we will receive something like "int variable equals two" in which the command is divided into 3 different sections: the type, the name and the value of the variable.

All this information is collected by manipulating the command string. The first step is to find the word "equals", because everything on the left is the type and name and on the right the expression of the variable.

The type is found in the first word of the command, because as explained in the "filtering procedure" section, commands that have separate words but together form a concatenated word, if this is contained in the list of possibilities, the separate words are removed and replaced by the concatenated word. This procedure ensures that the first word in these cases is always the type, making it easier to split the type and the variable name.

In the case of the variable name, it is everything between the "equals"("=") character and the first word of the command. If the name is made of more than one word the name will be written with the camelcase convention. The value of the variable is processed in a way that will be explained later for processing expressions.

The way mentioned above allows us to create an assignment statement using only a single command making this a very fast way that allows the user, if he already has the command in his head, to be able to "write" it very quickly, without having to resort to more than one command (see Figure 4.9).

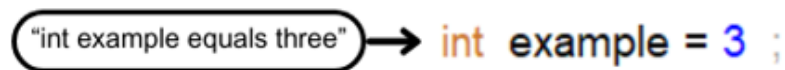


Figure 4.9. Command that creates an assignment statement with value.

Creating the statement in a direct way is not the only way, there is the possibility given to the user to create the statement in parts not being necessary to say the whole statement at once.

It is possible to create an assignment statement in which the name and expression are omitted, in which case a command such as just "int" will create a statement with the type referred to in the command and the name and value of the expression with spaces for completion (see Figure 4.10). This type of case occurs if the command consists of just one word and if that word belongs to the types found using AST, including primitive and non-primitive data types. If this hypothesis is validated, a statement is added to the AST with the type indicated and with a placeholder for an expression, which in the editor corresponds to whitespace.

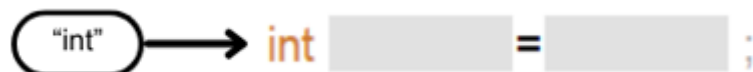


Figure 4.10. Command that creates an assignment statement just by saying the type.

With the same idea as above it was also made possible for the command to be composed only by the type and the name of the variable (see Figure 4.11). When this happens, the command does not

contain the word equals and the first word is a word representing a type. A statement is added to the AST in which the type and name is the one indicated in the command and an expression placeholder.

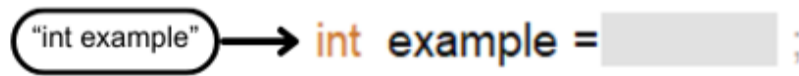


Figure 4.11. Command that creates an assignment statement just by saying the type and name.

As part of the expression statements, function calls can be used so it is also possible for the user to say for example the command "put number", in order to call a function called "putNumber" with an argument of integer type. After being validated by the list of possibilities, it is checked how many arguments are needed for the function and spaces are placed for the user in the following commands to put the information in place of the arguments (see Figure 4.12).

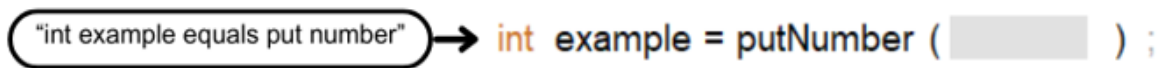


Figure 4.12. Command that creates an assignment statement with a function call.

In the case of the example we see that it contains one argument and so a space is placed for the user to decide what to put.

It is also possible to create return statement type commands using "return expression" type commands which when processed is divided into two parts. The reserved word "return" identifies the type of statement we want to create and the second part refers to the expression that is processed according to the expression part that will be explained later.

The editor also supports the control flow statements if-else, for and while. For both control flow constructs the creation process is quite simple, you just need to say the reserved word for the statement and automatically the structure is created. The control flow has different characteristics making the processing of the commands end up being different.

To create an if-else statement it is only necessary to say the reserved word "if" and is added to the AST a if statement with the expression with a blank space to be completed (see Figure 4.13).



Figure 4.13. Command that creates an if statement.

It is also possible to create a statement providing an expression in front of the reserved word, "if y equals 7" (see Figure 4.14). This case is processed by identifying that the first word is a word reserved for control flow types, and everything that follows is identified as an expression, adding to the AST an if statement with the expression of the command. This procedure is present in all control flow statements that have a condition.



Figure 4.14. Command that creates an if statement with an expression.

When you are inside the body of the if statement and you say the reserved word "else" the else statement is created (see Figure 4.15). This is possible because when we have the cursor in the if statement or within the if block, the word else is introduced into the list of possibilities, and when the command has "else" as its first word, an else statement is added to the if statement in question in the AST.

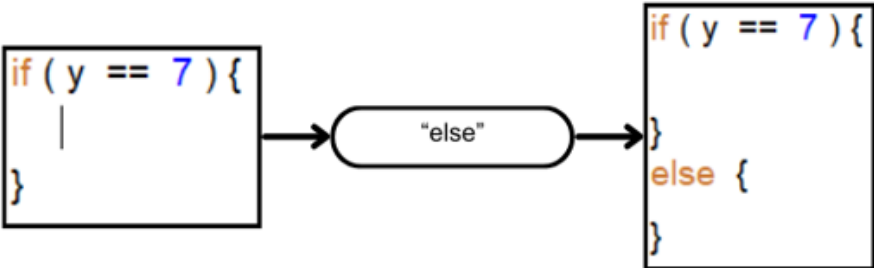


Figure 4.15. Command that creates an else statement.

To create a for statement, as mentioned above, it is only necessary to say the reserved word "for" and with this the structure referring to the control structure is created (see Figure 4.16).

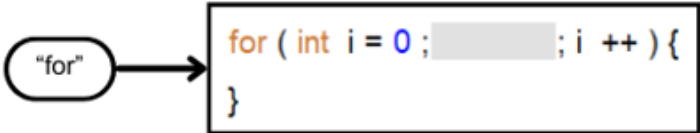


Figure 4.16. Command that creates a for statement.

Also as in the case above it is possible to use only one command to modify the expression of the control structure in this case "for y equals 7" (see Figure 4.17).

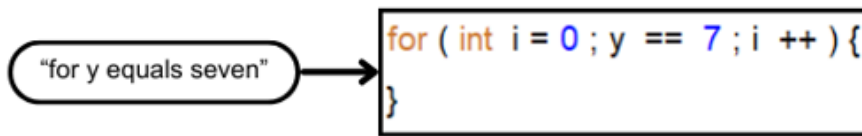


Figure 4.17. Command that creates a for statement with an expression.

To create a while statement it is only necessary to say the reserved word "while" thus resulting in the creation of the control structure and to change the expression of the while structure it is only necessary to say the expression in the command together with the reserved word in this case "while y equals 7" (see Figure 4.18).

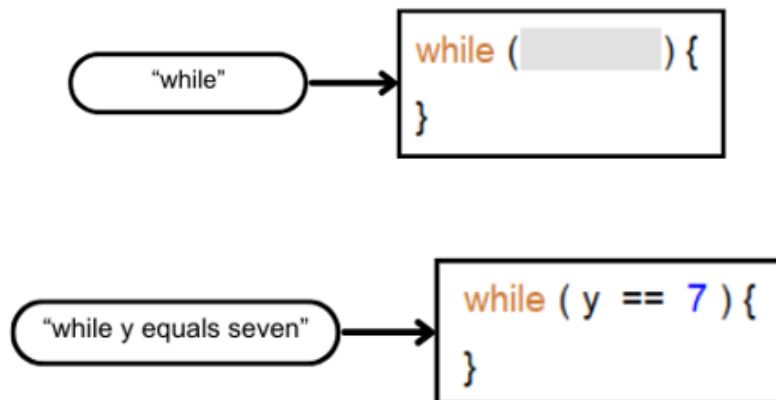


Figure 4.18. First a command that creates a while statement without an expression and below a command that creates a while statement with an expression.

4.2.9.2 Expressions

When we are with the cursor in a place where an expression is supposed to be, a certain freedom is given to the user. As explained a few chapters above, it is not mandatory to only use variables or values that are of the expected type at the end of the expression resolution. Giving a concrete example, if it were only possible to use variables, values or operators of the return type of the expression, it would not be possible to use integers in the condition of an if statement.

When an expression is processed, there is a step of substituting words that represent operators for the operators themselves, going from "greater than" to ">". Words that represent functions are modified by adding parentheses and the appropriate spaces for the number of arguments they have.

The most basic commands that can be made to manipulate expressions is to replace a certain value with something the user has said, which can be a variable or an arbitrary value.

If the cursor is over a value and the user says a value or variable, the previous value will be replaced by the new one said by the user.

When the cursor is in a place that contains nothing and is simply waiting for user input, the operator of the command is simply added creating a compound expression in the AST.

If the cursor is in a place where there is an expression, the operator is added as in the situation above but keeping the expression that was already there in the left branch of the compound expression (see Figure 4.19).

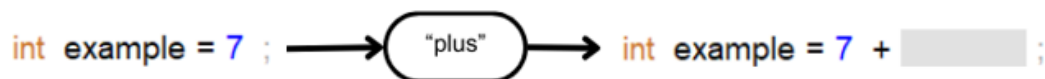


Figure 4.19. Creation of a compound expression with the first value already inserted.

It is also possible for the user to execute a command that is composed of a compound expression i.e. a value followed by an operator and again a value, if the cursor location is empty the expression is inserted in the location, if it is occupied the expression is replaced by the new one (see Figure 4.20).

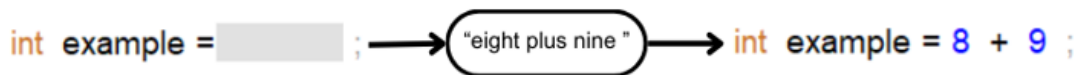


Figure 4.20. Creation of a complete compound expression.

Expressions can contain calls to functions within them, in which case when they are being processed the difference is made between whether the name is a variable or a function, as there is a list of function names created with the information from the AST. If the name is in the list of names, it is because it is a function call, in which case the parentheses are added and spaces are added inside them depending on the number of arguments it takes (see Figure 4.21).

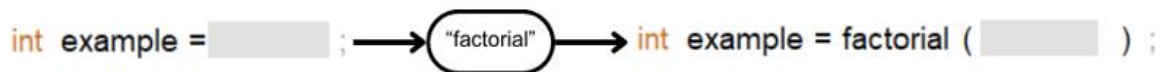


Figure 4.21. Creation of an expression with a function call inside.

All this type of expression manipulation consists of creating the expression within memory containing the changes the user wants and then replacing the bit of expression selected by the cursor in the AST by changing the model. The entire expression is taken into account so that nothing is lost during this replacement.

4.2.9.3 Types

When the cursor is on a type, either in a variable or in a method, it is possible to modify its type using commands with the reserved words for the types, whether these are primitive and non-primitive data types such as String and even classes that are in the project, for this the command is analyzed checking if it is a primitive or non-primitive type to then be replaced using the AST of the Javaparser changing the model.

In the case of arrays, an array is created when a command is made in which the type of the array is allowed, "int" for example, and the second word of the command is array followed by a name for the variable, "int array numbers".

5 Performance evaluation

This chapter evaluates the performance of the system prototype in two ways. Firstly, we propose a set of tests to be conducted by users on eight different tasks. Section 5.1 describes the tasks in detail, presents the achieved results with the tested users, and discusses the results. Section 5.2 presents an additional evaluation experiment, which compares the usability of speech commands and a structured editor with the usual keyboard and mouse.

5.1 User tests

User tests were carried out in order to test the user-friendliness, efficiency and usefulness of the prototype for end users. The tests consisted of tasks that include navigating the editor and modifying code using voice commands. Eight tasks have been defined, each of which requires the tester (experiment subject) to use commands to navigate the editor, along with commands to change the code. These tasks start out as basic so that the tester can explore and begin to understand how to interact with the editor.

5.1.1 Tasks definition

The following tasks increase in complexity to become as close as possible to a normal use of a code editor. With this in mind, the following tasks were created:

1. Create a field declaration with the name age of type int

In this task, the tester is asked to create a field declaration with a certain type and name for the class in which the editor is located. To do this, the tester is expected to place the cursor in the body of the class using voice navigation commands to move the cursor to the indicated place, and then create the field declaration with a voice command (see Figure 5.1).

```
int name ;
```

Figure 5.1. Task 1 objective.

The aim of this task is to find out how the tester adapts to using voice commands, and what they think of the command to create the field declaration, since they have to say the words "field declaration".

2. Create a private field declaration named id of type int

In this task, the tester is again asked to create a field declaration, but this time with the access modifier private. With this, the tester is expected to move the cursor back to the body of the class and use the command to create the field declaration again together with the access modifier (see Figure 5.2).

```
private int id ;
```

Figure 5.2. Task 2 objective.

The aim of this task is to find out how the tester finds it most intuitive to create the field declaration and what difficulties are encountered during creation.

3. Create a public method with named factorial

To do this, they need to use voice commands to move the cursor to the body of the class and then create the method with the name requested (see Figure 5.3).

```
public int factorial ( ) {  
  
}
```

Figure 5.3. Task 3 objective.

The aim of this task is to find out what the tester thinks of creating the method, as it is necessary to say the word "method".

4. In the test method add an int parameter named number

For this task, testers were asked to create a parameter with the characteristics requested above in a method called "test". To do this, you need to go to the body of the method or to the line of the method using the navigation commands and then use the command to add the parameter to the function (see Figure 5.4).

```
public int test ( int number ) {  
  
}
```

Figure 5.4. Task 4 objective.

The purpose of this task is to see how the tester adds the parameter and whether the way it is implemented is intuitive.

5. In the test method create an if to check when the number is greater than 18

For this task, the tester is asked to create an if statement with an expression. To do this, they need to move the cursor to the body of the method and then use the commands to create the control-flow statement indicated along with the expression (see Figure 5.5).

```
public int test ( int number ) {  
    if ( number > 18 ) {  
    }  
}
```

Figure 5.5. Task 5 objective

The aim is to see how the tester adds the control-flow statement in order to understand the most intuitive way for the command to behave.

6. Create a setter and getter for the field declaration age

The tester is asked to create a setter and getter method for a field declaration in the class. Again, the tester is expected to move the cursor to the body of the class, and then use voice commands to create the method of type setter and getter (see Figure 5.6).

```
public int getAge ( ) {  
    return this.age ;  
}  
  
public void setAge ( int age ) {  
    this.age = age ;  
}
```

Figure 5.6. Task 6 objective.

The purpose of this task is to understand how the tester creates the methods. There is also the possibility of creating these methods using specific commands that automatically build the methods.

7. Create a function called sum that receives two int parameters that gives the sum of those two numbers

For this task, the tester is asked to create a method that receives two integer arguments and sums them up. To do this, the tester needs to create the function, add two arguments and then return the sum of the two, creating a compound expression, always controlling the cursor with voice commands (see Figure 5.7).

```
public int sum ( int number , int otherNumber ) {  
    return number + otherNumber ;  
}
```

Figure 5.7. Task 7 objective.

The aim of this task is to get the tester to learn how to use compound expressions as it is necessary to use an operator together with the reserved word return.

8. Complete the factorial function

For this task, the tester is asked to complete the function created in task 4. To do this, they need to move the cursor to the body or line of the method to add a parameter of integer type to the function with a name of their choice, then move the cursor to the body of the method and complete the function, creating variables, flow-control statements, variable reassignments, and function calls as it is a recursive function so that at the end of the function the final result is the factorial of the number passed as a parameter (see Figure 5.8).

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

Figure 5.8. Task 8 objective.

The aim of this task is to analyze how the tester implements a complete method. The most difficult task is to understand which commands are less and more intuitive when creating a method.

During the tests, metrics were taken such as the duration of the task, the number of commands used to perform the task and whether the task was completed successfully.

The aim of collecting these metrics is that, with the duration of the task, we can see which tasks the testers spent the most time on, which will mean difficulty in completing the task. In the case of the number of commands, all those that the user says will be counted, even those that the editor does not support, and the number of commands said for the task will be compared to the optimal number of commands needed to complete the task. The high number of commands used may also be a reflection of the tester having difficulty completing the task.

5.1.2 Code writing performance results

This section presents the results with five tests, carried out on programmers, from the author's personal network. They were recruited by asking if it would be possible for them to test the prototype, and they aged between 23 and 35 who use code editors on a daily basis.

After completing the tests, the results were grouped together and the average time and commands used for each of the tasks were calculated. The testers' averages were compared with the time that the author of this dissertation took to complete the tasks plus a threshold, to simulate a non-biased regular user for which the tool is unfamiliar, and the number of commands used for each of the corresponding tasks. The number of commands used does not include editor navigation commands.

During the execution of the tests, the testers were asked to think aloud in order to take into account their thought process when carrying out the tasks. The task averages are shown in Table 5.1. Average times and commands used compared to the expected times and commands for each of the tasks.

Table 5.1. Average times and commands used compared to the expected times and commands for each of the tasks

Number	Task	Average duration	Expected duration	Average commands used	Expected commands
1	Create a field declaration with the name age of type int	01:02	00:25	4.6	1
2	Create a private field declaration named id of type integer	00:23	00:30	1	1

3	Create a public method named factorial that returns an integer	00:33	00:30	1.8	1
4	In the test method add an integer parameter with the name number	00:34	00:30	2	1
5	Create an if to check when the number is greater than 18	00:32	00:30	1.6	1 or 2 or 3
6	Create a setter and getter for the field declaration age	00:55	00:40	4.4	2
7	Create a function called sum that receives two int parameters that gives the sum of those two numbers	01:32	01:30	5.4	4
8	Complete the factorial function	02:08	02:40	7.6	5

5.1.3 Analysis of the results

Task 1 proved to be difficult for the testers due to the need to first say "field declaration" when creating the field declaration and only then the name of the field declaration itself. The users' first reaction was to say "int name", which is correct, but due to a point that was explained above in the notation of the method and the field declaration being the same, it is therefore necessary to add the keyword to distinguish the two cases.

Task 2 went quite well as it is identical to the one above except for the part about adding a visibility modifier and all the testers completed this task successfully without help.

For Task 3, most of the testers did not use the word "method" before saying the name of the specific method, which meant that it wasn't possible to create the method the first time around, but after nothing came up, they tried again with the word method as in task 1 and managed to complete the task successfully.

In Task 4, most of the testers again had a hard time figuring out that the word "add parameter" needed to be said before the notation of the parameter to be introduced, which meant that they needed help to add the parameter to the method.

One of the testers, on the other hand, tried to add the parameter to the method without the word with no success and explained that if in the others it was necessary to say the instruction first, in this one too it might be necessary, managing to complete the task without help.

In Task 5 we asked to create an if statement with a condition in which all the users managed to complete the task some with more commands than others by dividing the task into sub-tasks: creating

the if statement itself, and then adding the condition, but all the testers completed the task successfully without any help.

Task 6 is a task that in theory would require a lot from users and is much more difficult in terms of work because you have to create two methods and complete them, which seems like a big leap in complexity but in fact using the editor's shortcuts you only need 2 voice commands. The testers found it difficult to find the shortcut command to create the methods. Most of them tried to create the methods by hand, but with a little help with the setter command, the testers for the getter used the same reasoning, completing the task.

In Task 7, the testers were able to successfully create a method that performs the sum between the two arguments. The difference between some testers was that they created variables to store the sum value and then return that value, and others just returned the sum value directly on a single line, leading to more commands on the part of the testers who created the variable, but they all managed to create the method successfully and fairly quickly.

The last task was the most difficult and therefore the one that took the longest to complete, as it required creating control-flow statements, as well as using calls from the function itself, but it turned out quite well and all the testers were able to complete the task. The great challenge of this task was to create an else in which it is necessary to be on the line of the if or inside the if block, which took some time to figure it out for some of the testers.

The first tasks were the ones that were completed with help from the author of this dissertation, because it is difficult for testers to figure out that to create field declarations or methods, it is necessary to say the words "field declaration" and "method" in order for the creation to be completed. This was something that was mentioned by the testers as something that could be improved, the fact that it was not necessary to say these words would be something that would improve usability for users using the editor for the first time.

When adding a parameter to a method there is also an obligation to say "add parameter" which was commented on as an improvement to remove the need to say these words.

Another improvement mentioned by the testers was that the creation of the else statement didn't need to be inside the if statement block or on the line of the if statement itself in order to create the else statement, which caused a bit of trouble for some of the testers who tried to create the else statement outside of the two situations.

Positive feedback was given regarding the automatic creation of setters and getters with just one command instead of doing the methods by hand, which saves time when creating them. The navigation

system and the application itself was also something that users liked, describing how satisfying it is to watch the cursor change places and the code appear out of nowhere.

5.2 Jasay versus Keyboard and Mouse

A comparison has been made between using the keyboard and mouse and using voice commands, so that this is the only difference in the tests, which were carried out in Jasay solely by the author of this dissertation, to find out in which tasks one is better than the other in terms of finishing the task the fastest way.

5.2.1 Experiments and results

Different tasks were created with different objectives to try and cover different parts of editing code just like in the other test.

In order to understand which way might be more practical to use during situations, the execution time of the task was collected and then compared between the two modes of operation. To check that the times are valid, the cursor starts in the same position for each of the modes in each task, which will be the place where the previous task ends (see Table 5.2). The total time needed to complete all the tasks was also collected (see Table 5.3).

Table 5.2. Times made with the Jasay versus time made with the keyboard and mouse.

Number	Task	Time Jasay	Time Keyboard and Mouse
1	Create a field declaration named id of type integer	00:16	00:07
2	Create a public method named factorial that returns an integer	00:14	00:05
3	In the test method add an integer parameter with the name number	00:19	00:05
4	In the method test create an integer variable called newVariable	00:07	00:06
5	Create an if to check when the number is greater than 18	00:15	00:08
6	Create a setter and getter for the field declaration age	00:21	00:38
7	Create a function called sum that receives two int parameters that gives the sum of those two numbers	01:01	00:23
8	Complete the factorial function	01:17	00:41

Table 5.3. Overall times made with Jasay and Keyboard and mouse.

Mode Used	Time Overall
Jasay	03:50
Keyboard and Mouse	02:06

5.2.2 Analysis of the results

In general, the voice command mechanism was slower, requiring almost twice longer to complete all the tasks. This is due to the fact that the editor navigation mechanism is not the fastest because we are using voice and not something that places the cursor where we want it first, as the editor navigation is done by moving the cursor up, down, right or left. This makes it necessary to say more commands to get to where we want to make the changes. Bearing this in mind, it is normal that in most tasks, especially the first ones where you have to navigate the editor from place to place, the keyboard has a greater advantage in terms of time.

When we came to the task of creating setters and getters for a variable, the voice command mechanism managed to complete the task in less time than using the keyboard and mouse, due to the automation that exists when the intention to create functions of this type is detected.

Task 7 is the task where there was actually the biggest discrepancy in time, taking almost four times longer than using the keyboard and mouse, again largely due to the navigation not being as fast as with the keyboard and mouse.

Task 8 is the largest task in the tests in which the voice modification mechanism managed to keep up with the average time compared to the keyboard and mouse.

6. Conclusions and Future Work

With the rise of the software industry diseases that affect the joints due to repetition of certain movements (RSI) are present in everyday life. The creation of Jasay is intended to help those who suffer from these diseases. The interaction between the structured editor and the mechanism for modifying code using voice commands has been successful with the creation of Jasay, which is able to perform various functions of a programmer's day-to-day life, but using the voice. This type of system has the advantage that the changes made are at the level of the model, which ensures that it can be used with other types of structured editors because the editor is just one view of the model.

In response to the second research question, two types of tests were carried out: user tests and a time trial between Jasay and the keyboard and mouse in the structured editor. In the user test, positive feedback was obtained regarding the satisfaction of using the voice, but there was also negative feedback on aspects of using it. In the case of the comparison with keyboard and mouse, Jasay took almost twice as long to complete a certain number of tasks. Much of the time difference is due to the fact that navigation takes a long time compared to normal keyboard and mouse navigation.

Jasay is divided into two parts, the speech recognition part and the speech interpretation.

The speech interpretation is responsible for validating and executing the commands received by the speech recognizer. For the validation part it uses two strategies, the first one is based on a grammar in order for its rules to validate the commands, the second one consists in a list of possibilities containing language keywords and source code identifiers that are valid to use in expressions. The list of possibilities uses two different types of matching methods, the levenshtein distance and the phonetic distance, in order to approximate the speech recognition text and the set of valid instructions. If it passes the validation part, commands that modify the AST are formed and further applied.

A small scale experiment was also carried out to compare the use of the Jasay prototype to using the keyboard and mouse, the test consisting of completing certain tasks in the shortest possible time. These tasks cover various parts of the prototype, from creating field declarations, modifying methods....

The result turned out to be as expected: using the keyboard was quicker than using voice commands, but there are tasks that were quicker to do with voice mechanisms.

For future work, it would be interesting to review some implementation issues. The use of grammar in the case of adding field declarations and methods is good because the syntax when the field declaration has no expression is the same, but it was necessary to use a keyword to distinguish the two. The user tests showed that this implementation was not the most intuitive and could be improved trying to find a solution for this problem.

When a statement is created in which the value of the expression is omitted, for example, "int number", "int number = #HOLE;" is automatically created, so that cases such as "int number;" cannot be created, and it is interesting that the user can choose between the two possibilities.

Expand by creating more functionality for the application, for example use import functions, use functions corresponding to the type of variable being used, in this case if we have a String we can use the "equals" function.

The biggest difference between times was in navigating the editor. Although navigation has been improved by being able to repeat the same movement several times at once, combining an eye-tracker to control the cursor would be something to experiment, as it would significantly make navigating the editor much faster. Eye-tracking integration in Javardise has been experimented previously [18], and hence, in principle the two techniques can be used in combination. The use of an eye-tracker could be the solution to some of the problems mentioned above where extra user feedback is needed to help the prototype really understand the user's intention. This can be useful in solving the two problems listed above, helping to decide which statement to use and deciding which structure the user wants to create, a field declaration or a method.

The creation of Jasay and the results of the tests prove that the concept of a structured code editor with mechanisms for changing code by voice is a help for people who may suffer from conditions such as RSI, and may in the future, with the necessary improvements, be something that can really go hand in hand with a normal IDE in which the keyboard and mouse are used.

6 References

- [1] Rianto, A. Hermawan and P. I. Santosa.(2018). "Knowledge and Prevention of Repetitive Strain Injury Among Computer Users," *2018 International Conference on Orange Technologies (ICOT)*, Nusa Dua, Bali, Indonesia, 2018, pp. 1-4, doi: 10.1109/ICOT.2018.8705901.
- [2] Baba, N.H., & Daruis, D.D. (2016). Repetitive strain injury (rsi) among computer users: a case study In telecommunication company. *Malaysian Journal of Public Health Medicine*, 16.
- [3] Désilets, A., Fox, D. C., & Norton, S. (2006). VoiceCode: An innovative speech interface for programming-by-voice. *Conference on Human Factors in Computing Systems - Proceedings*, 239–242. <https://doi.org/10.1145/1125451.1125502>
- [4] Desilets, A. VoiceGrip: A Tool for Programming-by-Voice. *International Journal of Speech Technology* 4, 103–116 (2001). <https://doi.org/10.1023/A:1011323308477>
- [5] - Gonze, Damien. *Coding with the voice*. Ecole polytechnique de Louvain, Université catholique de Louvain, 2020. Prom. : Bonaventure, Olivier. <http://hdl.handle.net/2078.1/thesis:23049>
- [6] Rosenblatt L. (2017). VocalIDE: An IDE for Programming via Speech Recognition. In Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '17). Association for Computing Machinery, New York, NY, USA, 417–418. <https://doi.org/10.1145/3132525.3134824>
- [7] Ribeiro P, Silva D, De Janeiro R. (2021). Ferramenta de tecnologia assistiva baseada em voz capaz de ajudar programadores a programar na linguagem Javascript. Centro Universitário Carioca Curso de Ciência da Computação.
- [8] Nizzad, A. R. M., & Thelijjagoda, S. (2022). Designing of a Voice-Based Programming IDE for Source Code Generation: A Machine Learning Approach. *Proceedings - International Research Conference on Smart Computing and Systems Engineering, SCSE 2022*, 14–21. <https://doi.org/10.1109/SCSE56529.2022.9905095>
- [9] Hossain, S., Emi, M. A., Mishu, M. H., Zannat, R., & Ohidujjaman. (2021). Code Generator based on Voice Command for Multiple Programming Language. *2021 12th International Conference on Computing Communication and Networking Technologies, ICCCNT 2021*. <https://doi.org/10.1109/ICCCNT51525.2021.9579880>

- [10] Shahane, A., Kahate, A., Gorad, T., & Sonawane, P. (2019). V-IDE: Voice controlled IDE using Natural Language Processing and Artificial Intelligence. *International Research Journal of Engineering and Technology*, 195.
- [11] Paudyal, B., Creed, C., Frutos-Pascual, M., & Williams, I. (2020). Voiceye: A multimodal inclusive development environment. *DIS 2020 - Proceedings of the 2020 ACM Designing Interactive Systems Conference*, 21–33. <https://doi.org/10.1145/3357236.3395553>
- [12] Zan, T., & Hu, Z. (2023). VoiceJava: A Syntax-Directed Voice Programming Language for Java. *Electronics (Switzerland)*, 12(1). <https://doi.org/10.3390/electronics12010250>
- [13] Këpuska, V. (2017). Comparing Speech Recognition Systems (Microsoft API, Google API And CMU Sphinx). *International Journal of Engineering Research and Applications*, 07(03), 20–24. <https://doi.org/10.9790/9622-0703022024>
- [14] Radford, A., Kim, J. W., Xu, T., Brockman, G., Mcleavey, C., & Sutskever, I. (2022). *Robust Speech Recognition via Large-Scale Weak Supervision*. <https://github.com/openai/>, <https://doi.org/10.48550/arXiv.2212.04356>
- [15] McNutt, A. and Chugh, R. (2023). Projectional Editors for JSON-Based DSLs. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, <https://doi.org/10.48550/arXiv.2307.11260>
- [16] André L. Santos. (2020). Javardise: a structured code editor for programming pedagogy in Java. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming (Programming '20)*. Association for Computing Machinery, New York, NY, USA, 120–125. <https://doi.org/10.1145/3397537.3397561>
- [17] Aidan Pine, Patrick William Littell, Eric Joanis, David Huggins-Daines, Christopher Cox, Fineen Davis, Eddie Antonio Santos, Shankhalika Srikanth, Delasie Torkornoo, and Sabrina Yu. (2022). Gi22Pi Rule-based, index-preserving grapheme-to-phoneme transformations Rule-based, index-preserving grapheme-to-phoneme transformations. In *Proceedings of the Fifth Workshop on the Use of Computational Methods in the Study of Endangered Languages*, pages 52–60, Dublin, Ireland. Association for Computational Linguistics.
- [18] André L. Santos. (2021). Javardeye: Gaze Input for Cursor Control in a Structured Editor. In *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming (Programming '21)*. <https://doi.org/10.1145/3464432.3464435>