CodeSpeech: An Eclipse Plugin for Java Programming by Voice

Łukasz Strzelecki

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Mobile Computing

in Hagenberg

im August 2019

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, August 26, 2019

Łukasz Strzelecki

Contents

Declaration iii

Abstract vii

1 Introduction 1
- 1.1 Problem 1
- 1.2 Motivation 2
- 1.3 Goal 3
- 1.4 Overview 3

2 Related work 5
- 2.1 VoiceGrip & VoiceCode 5
- 2.2 VocalProgrammer 6
- 2.3 VocalIDE 6
- 2.4 SPEED: Speech Editor 7
- 2.5 NaturalJava 8
- 2.6 CodeTalker 8
- 2.7 Vocal 9
- 2.8 Voice and Gesture Development Environment 9
- 2.9 Voicecode.io 10
- 2.10 SOPE and ELDI 10
- 2.11 Problem of speaking code 11
- 2.12 Summary 11

3 Tools 13
- 3.1 Eclipse 13
- 3.2 Eclipse PDE 14
  - 3.2.1 Necessary software 15
  - 3.2.2 Project creation 15
  - 3.2.3 Adding extensions 16
- 3.3 Speech Recognition tools 18
  - 3.3.1 Explaining speech recognition 18
  - 3.3.2 CMUSphinx 20
  - 3.3.3 Google Cloud Speech-to-Text 22
  - 3.3.4 Comparison of SR tools 22
- 3.4 ANTLR 24

3.4.1 Typical workflow 24
3.4.2 ANLTR plugin 28

4 Method 29
4.1 Requirements 29
4.1.1 Functional requirements 29
4.1.2 Non-functional requirements 30
4.1.3 Constraints 30
4.2 Use cases 31
4.3 Design 33
4.3.1 System Architecture 33
4.3.2 Interface 35

5 Implementation 38
5.1 Packages 38
5.1.1 *general* package 38
5.1.2 *plugin* package 39
5.1.3 *speechrecognition* package 39
5.1.4 *interpreter* package 39
5.2 Most important classes and interfaces 40
5.2.1 PluginManager 40
5.2.2 SpeechRecognizer 40
5.2.3 RecognizerThread 40
5.2.4 Interpreter 42
5.2.5 InterpreterContext 42
5.2.6 BaseKeywordListener 42
5.2.7 Command 43
5.2.8 Operations 43
5.2.9 Models 43
5.3 Workflow 43
5.3.1 Speech recognition 44
5.3.2 Interpretation 46
5.3.3 Performing operation 47
5.4 Implemented features 47
5.4.1 Error Correction 47
5.4.2 Free Speech Mode 47
5.5 Addition of new functionality 50

6 Evaluation 55
6.1 Approach 55
6.2 Results 58
6.3 Discussion 59
6.3.1 Programming time 59
6.3.2 Interaction with mouse and keyboard 60
6.3.3 CER 60
6.4 Ideas for improvement 60
6.4.1 Interpretation of more propositions 61

6.4.2 Context recognition . . . 61
6.4.3 Combining recognition systems . . . 62
6.4.4 Better error correction . . . 62
6.4.5 Adaptation . . . 63
6.4.6 Feedback . . . 63
6.4.7 Improve voice control experience . . . 63
6.4.8 More features . . . 64

7 Conclusions 65
7.1 Why programming by voice . . . 65
7.2 Summary of the development process . . . 65
7.2.1 Tools selection . . . 66
7.2.2 Architecture design . . . 67
7.2.3 Implementation . . . 68
7.3 Conclusions . . . 69
7.4 Future work . . . 69

A CD-ROM/DVD Contents 71

B Source Code 72

References 75
Literature . . . 75
Online sources . . . 77

Abstract

As other computer users, some of the programmers develop Repetitive Strain Injuries (RSI) or for other reasons have troubles with using keyboard and mouse. Not to mention people whose disabilities prevent them from taking up a work as a programmer in the first place. With Speech Recognition (SR) systems getting better, another way of interacting with computers opens up. Programming, however, because of its specific syntax cannot be successfully done using available, commercial SR tools. Therefore an idea for a new system for programming by voice arose. That gave birth to a prototype called CodeSpeech, developed as an Eclipse IDE plugin for Java programming language created especially for this purpose. This paper deals with the whole process behind its creation, including requirements definition, design of architecture, description of utilized tools, implementation phase and ends with a first evaluation of the initial iteration's product, deriving conclusions, discussing them and proposing certain solutions for future development. The evaluation compares the new programming method using CodeSpeech with the standard way of programming using mouse and keyboard by recreating a given sample program and measures the completion time, as well as the use of peripheral devices. In addition, Command Error Rate (CER) metric was measured during the test. The conducted experiment showed that the prototype is around four times slower than the regular programming method, it decreases however the time of mouse and keyboard usage to 37,6%, distance traveled by the cursor to 17,4%, keystrokes required to 29,9% and the number of clicks and doubleclicks to 20% and 5% respectively. Calculated CER was 29,5%, meaning there is room for improvement. The result shows that in the current state CodeSpeech is not feasible to replace mouse and keyboard due to the time it requires for program completion, nonetheless it shows that the potential exists, and with further program development a time might come when its performance becomes competitive to the standard means of programming while keeping all of the benefits of contactless programming.

Chapter 1

Introduction

This thesis deals with a CodeSpeech project - an Eclipse IDE plugin whose goal is to present an alternative for standard means of programming, which is nowadays exclusively done via mouse and keyboard. The prototype allows using programmer's own voice in order to create basic programming structures in Java language and perform basic operations in the integrated development environment. The paper starts with an explanation of motivation standing behind the idea of introduction of such tool and describes the problem that it tries to solve. Then the document continues with a presentation of a whole thinking, creative and practical process of architecture design and implementation phases, followed by system's evaluation and discussion of results to finally finish with a summary of what has been done, what are the lessons learned throughout the whole experience, stating arising conclusions and potential enhancement possibilities to be performed in the future.

1.1 Problem

Computer science studies (or related topics), as the name suggests, involve the use of computers. This includes an enormous amount of hours spent sitting in front of a desk. According to research done by Schlossberg et al. on engineering graduate students,

> During the first year, 80% used a computer more than 20 hours per week and by the fifth year this had increased to 95%. Those reporting greater than 40 hours of computer use per week increased each year in graduate school, beginning with 34% of participants in the first year and increasing to 56% in the final years of study. [17]

Very often during the late period of studies, an internship or a temporary job in the field is taken up by a student which increases the amount of time spent that way even further.

Although results of existing studies on direct relation between usage of computer and severe health conditions vary, many of them involve long term computer users that report pain. Schlossberg et al. confirm in their research done on randomly sampled 206 Electrical Engineering and Computer Science graduate students that *hours of computer use per week is associated with an increased risk of persistent or recurrent upper extremity or neck pain* [17]. Lassen et al. performed a year lasting study on 6943 computer operators which ended in a positive association between self-reported mouse and keyboard times and elbow and wrist/hand pain [13]. Their later study, however, did not confirm that the use of mouse and keyboard are predictors for persistent and severe pain [14]. Chang et al. state that their data *suggest a potential dose–response relationship between daily computer usage time and musculoskeletal symptoms* [8]. Brandt et al. conclude their study with the statement that *mouse use is associated with an increased risk of moderate-to-severe pain in the neck and right shoulder, and an association with tension neck syndrome is possible* [7].

Many of the problems are suspected to be caused by repetitive movements that working with peripheral devices such as mouse and keyboard requires. These symptoms started to be known under an umbrella term “Repetitive Strain Injuries” (RSI). This topic is broadly described by Van Tulder et al. [22] and Yassi [24]. Another factor that supposedly has a negative effect on health is sedentary lifestyle, inevitable in any kind of desk work (so also with computers). Tremblay et al. analyse this problem deeper [20]. Van den Heuvel et al. [21] and Walker-Bone et al. [23] prove that physical injuries can result such negative consequences as increased amount of absence from work as well as decreased productivity at work. Aforementioned issues do not touch only computer scientists, but any profession that is the use of computers on a daily basis such as office workers, graphic designers etc. Main problems that are frequently reappearing are neck, upper extremity pain (shoulder, elbow, wrist and hand), Carpal tunnel syndrome, lower back pain and similar musculoskeletal symptoms.

1.2 Motivation

The author of this thesis himself fell victim to some of those issues and after many years of studies developed chronic pain in lower back, neck and recently also shoulders. The pain began to be very problematic not only in times of working on a computer, but also in everyday life. That was the main reason for the idea of creating a new way of interaction with computers with emphasis on programming, which is an occupation of the author.

Progress in certain fields of science such as Machine Learning and Big Data opens up new possibilities that previously could only be imagined. Speech Recognition (SR) is one of the most “recent” technologies that benefit from these fields of science and start to very popular. Nowadays SR is used by many and it takes many forms. Be it Amazon’s Alexa, Apple’s Siri, Microsoft’s Cortana, Google’s Assistant and more, they all allow for voice control over devices. Speech Recognition is being utilized in smartphones, cars and automated households. It is this technology, that the author saw as the one that brings in a new opportunity for alternate way of programming. Unfortunately, available commercial tools are not suitable for this task. Those are mainly created to write text in national languages and perhaps to recognize special commands for system control. Programming languages, however, because of their specific syntax cannot be successfully transcribed by them. The grammar of many programming languages requires awkward use of brackets, parentheses, semi-colons and other punctuations. Therefore a need for an adhoc solution appeared, a tool that could solve the problem of awkward code formatting and enable programming by voice.

Such a tool would not only help to prevent the development of upper extremities pain linked to the use of mouse and keyboard, but also free the programmer from the need of sitting in front of the desk. With nowadays broadly available wireless microphones, the potential software developer could move away from a workstation and take more comfortable position, perform stretches, simple exercises or just walk around the room giving them the opportunity for much more active and healthier lifestyle than it has been possible up to this point.

Going further with this idea, another potential usage was found. There are people with more severe physical conditions than just chronic pain, namely upper body impairments or a paralysis. The state of such people obviously makes it impossible for them to use a standard way of interacting with computers and thus excludes them from taking up a potential occupation as a programmer. Different tools were created to allow disabled people to utilize computers, none yet specifically designed for coding. How great would it be to enable some people with brilliant minds yet limited in their bodies to take a job as a programmer and to be as productive as completely healthy person. This could improve their lives in many ways. It could give them purpose and a way of greatly contributing to the society by development of the new software. That obviously would lead to the improvement of their material situation, in result making it easier to pay for many specific needs these people have.

It seems that, even though such a tool might not have a big impact on the world, it still can prove useful to many existing programmers suffering from occupational injuries and at the same time provide a new group of programmers arising from physically impaired people to fill the ever growing needs of the market.

1.3 Goal

The goal of the project is to create a software that will allow programming using voice. The tool should be somehow compatible with an already existing Integrated Development Environment (IDE) and should utilize available speech recognition engine. Creating new, specific SR engine or programming environment from scratch are not the topic of this thesis, but rather combining different, already existing tools and forming a completely new product.

1.4 Overview

This section gives an overview of the content of this thesis by briefly describing each subsequent chapter and what can be found in it.

The second chapter titled *Related work* focuses on description of several existing studies related to the topic of voice programming. Most of the selected documents present similar programming by voice projects. These projects are briefly presented in this chapter and compared to the subject of this thesis. Additionally, other research that is not based on any system but is still relevant is mentioned.

The third chapter is titled *Tools* and there one can find description of various software that have been used in order to complete the CodeSpeech project. The first section is dedicated to Eclipse IDE, the second to its extension for plugin development, the third one is focused on three different speech recognition technologies: belonging to

CMUSphinx family of open-source tools PocketSphinx and Sphinx4, and a commercial Google Cloud Speech-to-Text service. The end of this section deals with performance comparison thereof. The last section describes a parser software called ANTLR.

Right after that there is *Method* chapter, which is a place where functional and non-functional requirements are stated, use cases defined and the design of architecture is presented.

The *Implementation* chapter as the name suggests is fully devoted to the process of implementation. It describes the structure of the project based on previously defined architecture, presents the most important classes and interfaces, guides through the workflow of a working cycle to finally discuss some of the interesting features and walk through the process of adding new functionality backed by an example.

The next on the list, *Evaluation* chapter deals with the test that was undertaken on the finished prototype. The approach is being described in detail to leave no doubts about how it was performed. The metrics of interest are explained together with the way they were recorded. Then there is a presentation of the results followed by the discussion of the received scores and what do they mean. In the end, a couple of ideas that arose during or after the evaluation as to how the project can be improved are presented.

Finally, the *Conclusions* chapter summarizes what has been done up to this point. It goes through the statement of the initial problem and how the proposed solution potentially solves it. Then it describes a way the goal was achieved with thinking, creative and practical process behind it, as well as the experience gained throughout the course of development. An objective evaluation of the system overall, how well does it serve its purpose and satisfies initial requirements is given. The chapter and therefore the whole thesis concludes with the statement on the future plans for the project and its continuation.

Chapter 2

Related work

This chapter focuses on a short summary of similar projects that have been done in the past and comparing them with the one that is the main topic of this paper. After reading this chapter it should be clear how this project is different from others, what is innovative and what is done in a similar matter. At the end, other important studies related to the topic of programming by voice are discussed.

2.1 VoiceGrip & VoiceCode

Désilets et al. describe VoiceCode as “Innovative Speech Interface for Programming-by-Voice” [10]. After reading their paper and watching an online video presenting its functionality [32] it does seem that this is the most complete software in comparison with others. It is not a surprise, seeing how it has been developed through many years, evolving from earlier VoiceGrip project described by Désilets [9]. Désilets et al. see the problem in the use of existing SR tools for the purpose of writing code and put focus on providing a way to use more natural utterances. The earlier document has more content and goes through the tasks with which programmers have to deal with in their everyday job and what problems occur in performing them while using SR. VoiceGrip already handles well many of those tasks. The main difference between the two, is that in VoiceGrip, so called pseudo code is being written down word by word and is translated to native code by the Code Translator only when programmer runs it. In VoiceCode the translation is done in real time. Another big modification introduced in the new version was an addition of error correction. It provides two types of error correction: **Not What I Said** - when VoiceCode did not recognize the utterance properly and **Not What I Meant** - to address situations where recognition was correct but translated to the wrong code. This two fold system does not only allow for an easy modification of a mistake but also lets the system to adapt and avoid same mistakes in the future. Both projects allow code navigation. Because it is not clearly stated in what form VoiceCode comes, it is assumed that it is a similar tool as its predecessor, meaning that it can be utilized with different SR systems, such as Dragon NaturallySpeaking, and it can be used by different programming editors, like e.g. Emacs. It requires, however, an application of few macros before use. CodeSpeech on the other hand was built as a plugin to commonly used, free of charge IDE - Eclipse and uses built-in SR. It is therefore neither cross SR, nor cross editor. This can come as an advantage or disadvantage with the relation to the other two. First, it is limited only to one IDE and thus cannot be used with another one, possibly preferred by a potential programmer. On the other hand, no special configuration is needed and once the program with plugin is installed, it enables programming by voice with a single click. Another advantage over the aforementioned tools, coming from the choice of a specific, broadly developed IDE is that it is easy to enhance CodeSpeech to execute every single command that exists in Eclipse environment, such as adding breakpoints, compilation, debugging etc., as well as calling functionality from other plugins, if those are available in the current instance. The software that is the topic of this thesis is just a prototype, therefore it lacks many of the important features that the programs developed by Désilets and the team have, such as sophisticated error correction or transferability across programming languages (VoiceCode can enter specific syntax depending on file extension while CodeSpeech is strictly developed for Java). Many of the provided functionality like local navigation, is still implemented in a limited manner. By the time of writing this thesis neither VoiceGrip nor VoiceCode were available online to try them out.

2.2 VocalProgrammer

Arnold et al. [1] have designed a generator of voice recognition syntax-directed programming environments called VocalGenerator. It takes as input context-free grammar for a programming language and a voice vocabulary for that language. Because of that it supports different languages such as Basic, C, C++, Java etc. The authors argue that because of its syntax-directed editing two major capabilities are provided that the standard text editors do not include: navigation and selection. Having that, the authors decided to create another project based on VocalGenerator called VocalProgrammer. This project was supposed to be using using Dragon NaturallySpeaking and Microsoft Visual C++ with focus on Java programming language because its full documentation is available online for free. This tool would require a user to have purchased Dragon NaturallySpeaking Professional Version. Because of no further mention of this project it is assumed that it has never came to a conclusion.

VocalGenerator is a completely different tool than CodeSpeech. Deriving from description, however, it seems like a similar tool could be developed by it. VocalProgrammer was supposed to be such a tool. It was to be based on Java similarly to CodeSpeech. CodeSpeech does not require any external SR engine as VocalProgrammer would. VocalProgrammer lacks modularity that Eclipse provides, therefore scalability of its functionality seems limited. VocalGenerator allows creation of many editors for different languages, which is a big advantage.

2.3 VocalIDE

VocalIDE is a prototype created in JavaScript created by Rosenblatt et al. [16]. It is a web application that allows writing source code using a set of vocal commands. It records and recognizes users' speech using WebkitSpeechRecognition currently available in browsers such as e.g. Google Chrome, and rule-based syntax parser. It allows text entry, navigation, text selection, replacement, deletion and snippet entry. Text entry works like in a regular speech recognition program meaning each symbol character has to be explicitly given ("type open parenthesis i space less than space one close parenthesis to enter (i < 1)"). It makes it rather inconvenient to program by voice and is something that CodeSpeech successfully avoided. It is an interesting idea to create such a tool as a web application, which has an advantage of the fact that it can be accessed from everywhere where any kind of computer with a browser is available (which nowadays is rather taken for granted). Requirement for the Internet can be a disadvantage at the same time. CodeSpeech is strictly dependent on a specific program (Eclipse), which has to be installed on a computer first. As for Internet access, for now it is a requirement since it is preferred to use Google Cloud Service, which provides better recognition. Both programs are based on rule-based syntax parser.

Local navigation in both projects is done in a similar manner - the user can say "go to line" with a number to jump to a specific line. VocalIDE has developed text selection by word as well as replacement which is something that CodeSpeech lacks so far. Undo operation and smart snippet entry (entering ready code block) are both available in CodeSpeech. Interesting idea presented in this paper is Context Color Editing. Instead of moving the cursor to left or right by one character, nearby words are highlighted in different colors. Saying that color's name moves the active selection to the respective word.

VocalIDE was developed based on the results of research that used Wizard of Oz method. The system was evaluated during the study performed on eight impaired participants. Both researches, as well as the design of a prototype were mostly focused on navigation and selection in Java code.

2.4 SPEED: Speech Editor

SPEED is a program editor described by Begel [2] which together with program analysis framework called Harmonia [5] creates a system embedded in Eclipse IDE. It allows for code creation and edition together with high-level program manipulations like refactoring. To translate voice commands to code it uses SpokenJava - a semantically isomorphic dialect of Java that is more natural to be verbalized [3]. It allows to replace phrases like "open braces" with "begin class" in order to create an open bracket after class declaration. That seems to be a big difference between CodeSpeech and SPED, namely, CodeSpeech uses context-free grammar, which once recognized, creates a specific programming construct without the need of handling parenthesis, brackets or explicit starting or ending of these constructs. In order to help people with voice input Begel's project mentioned adding a specific spoken feedback system that would make its use easier to speak proper phrases. Because of strict dependency on grammar, CodeSpeech requires a similar feedback system to help users in learning of the commands for smooth usage. Additionally, a special search and selection tool was being developed for SPED, to help with finding a specific words based on phonetics without spelling a word precisely - similar system is needed for CodeSpeech.

The SPEED system has quite sophisticated feature to handle and support ambiguities through the syntactic phases of program analysis, as well as support for combination of commands and code. It is not clear at which stage of development the project has ended because no further studies on it were found. Just like the rest of the above mentioned projects, this one was not available at the time of writing the thesis.

2.5 NaturalJava

Shinde describes an interface for programming called NaturalJava [18]. It allows entering English sentences as input which produce Java source code as output. It uses information extraction techniques to generate so called case frames that classify key concepts of the sentence. The case frames are then used to modify Abstract Syntax Tree (AST) accordingly. This is the first project (from above mentioned) that uses manipulation of AST rather than being simply text based. Same method is used in CodeSpeech, where specific commands result in creation, deletion or modification of AST nodes. NaturalJava has a very interesting way of interpreting English sentences, which seems to get triggered based on detected keywords such as "iterates" in sentence "create a for loop that iterates from 1 to 5". CodeSpeech analyses provided utterances via parsed tree walking, and similarly triggers specific actions based on detected keywords. The biggest difference is that NaturalJava does not utilize any speech recognition tool, utterances are to be typed in order to generate code.

2.6 CodeTalker

Snell and Cunningham propose CodeTalker project[19]. The program was developed as a standalone editor that allowed simple source code creation (in Java) such as definition of a class, function, variable, constructs like for, while loops and switch statements. The editor was created very carefully, iteratively with each version being evaluated in terms of Human Computer Interactions (HCI). The paper also mentions the possibility of creating HTML code. It contains many useful features, such as replacement of previously defined spoken keywords with text (e.g. saying class will insert `class { \n }`) and addition of completely new, personal replacements via dialog wizard. It also has an option to highlight syntax, as well as to add specified keywords to be highlighted. Example of spoken input is "public class my class, braces, private void my function, open brackets, close brackets, braces, count equals my variable, semi-colon". As can be seen, the punctuation has to be explicitly stated which is something that CodeSpeech wanted to avoid by all means. This, however, allows the CodeTalker to detect if a construct is a class, a variable or a function. In many situations the end of a spoken utterance has to be marked by saying either "tilde" to invoke command or 'circumflex" to cancel it. CodeTalker provides a way for entering a word that is reserved as a keyword, which is a common problem in command based systems. Main differences are dependency on other tools and functionality. While CodeSpeech is a plugin that obviously requires Eclipse IDE, CodeTalker is a standalone tool. It requires, however, an external speech recognition software. The former makes operations on AST rather than text, and allows more operations than just creating constructs with specific names but also provides a way to utilize built into Eclipse commands that are important part of the programming, such as compilation and debugging.

2.7 Vocal

An interesting idea of bringing programming by voice to smartphones was brought by Islam et al. [12]. "Vocal - a voice based code generator" is an Android application that enables creation of code by voice. It uses Google Cloud Speech API (similarly to CodeSpeech) as it turned out to be the most accurate from total of free that they have tested out. Only creation of simple structures such as integer or character variables, arrays, method definitions, while and for loops and printing to console is possible in this program. The generated source code is of Java programming language. This software requires an Internet connection to exchange data with the cloud for the feature of compilation on the server. Database is used for storing text files. Although mobiles are not yet suitable for programming, the idea itself is curious. The use of smart devices could potentially make it possible to program conveniently in any place and taking into account Internet accessibility in the world it sounds feasible. Obvious difference between this project and the CodeSpeech is targeted device.

2.8 Voice and Gesture Development Environment

Voice and Gesture Development Environment (VGDE) is an interesting project presented by Bouse [6]. It combines together an idea of voice and hand gesture control of the system. Voice commands are used to create and edit code, navigate through it and interact with the program. Gesture commands on the other hand are used only for editing and navigation in the file. This provides a possibility of mixing two methods together as they run in concurrent threads. The author recommends that to achieve the best results one to two second breaks should be done between the commands. This probably makes programming with VDGE quite slow, while in given example "Bar Equals One Okay" there are four commands, therefore four breaks are needed. This commands would produce code "bar = 1;". The evaluation of VDGE, however, did not measure the system's execution time. To be fair, CodeSpeech also requires around two seconds break between commands as this is the average time of recognition. The difference is, that in the example mentioned above VDGE's four commands correspond to one command in CodeSpeech.
Voice recognition is performed by Microsoft Speech and gesture recognition requires external device, namely Leap Motion. Leap Motion is a small device that can be placed between the user and a keyboard. It tracks hands with discrimination of fingers. VDGE unlike CodeSpeech is a standalone development software. Its code manipulation is done on text and does not perform AST operations as CodeSpeech does. It does, however, allow for more detailed navigation in the code. VDGE's targeted programming language is Java, as well as it is of CodeSpeech. Commands are based on grammar and parsed in a similar manner in both programs. Both tools are limited as to the complexity of program that can be created using them, although they do possess the potential for extension.

2.9 Voicecode.io

At the time of writing the thesis one of the most recent programming by voice tools being created was Voicecode.io. The author of the tool is Ben Meyers. The tool on the official website is described as

> a concise spoken language that controls your computer in real-time. When writing anything from emails to kernel code, to switching applications or navigating Photoshop - VoiceCode does the job faster and easier.
>
> VoiceCode is different from other voice-command solutions in that commands can be chained and nested in any combination, allowing complex actions to be performed by a single spoken phrase [33].

From the description it can be concluded that the tool is not a programming tool per se but a general software for system control and it is de facto that. To allow system control Voicecoe.io integrates Dragon NatruallySpeaking SR technology and SmartNav for mouse-free control of the cursor. SmartNav uses a small device that sends infrared waves which reflect from a small pin that can be attached to the baseball cap. This allows for tracking position of the head and moves the cursor accordingly. It is a similar solution to eye tracking technology. The general idea for hand-less system control is great by itself, however what is at the point of interest for the sake of this thesis is its ability to write code. On one of the videos available on official website there is a presentation of an example. The code being written is in JavaScript. Written code is only three lines long and is being written fairly fast. Unfortunately, commands that are spoken to achieve this goal sound nothing like natural English sentences. It is hard to follow and understand which command does what and probably a lot of learning has to be done in order efficient. There is a second video showing writing a script in Ruby, meaning that Voicecode.io is not limited to one programming language, perhaps because it simply writes dictated text (so also punctuations). Currently Voicecode.io supports only MacOS and is constantly under development.

2.10 SOPE and ELDI

An interesting paper written by Bettini and Chin [4] comes from year 1990. They describe inside the first results from a research project on man-machine interaction based on speech. SOPE (Speech Oriented Programming Environment) was a project of which the goal was to create programming interface steered by voice. At the beginning the main focus was put on debugging process. In order to do that, IBM's Tangora speech recognition was used and created by them ELDI (English Language Debugger Interface) prototype based on Augmented Transition Networks (ATN). Their goal was to use natural English sentences based on grammar, which at the beginning was quite strict. The restricted vocabulary was up to 60-70 entries. The basic concept they presented is very similar to current solutions, yet at that time as the authors conclude, speech recognition technology was not good enough for this kind of tool to be productive.

2.11 Problem of speaking code

Begel [2], Arnold et al. [1] , Désilets [9] and authors of other previously mentioned papers described problems related to the programming by voice. One of the biggest issues is the fact, that source code is known only in written form and there is no unified way to read it. Every programmer, when asked to read code out loud, has a specific way doing so. Analogically the same problem occurs when the program is supposed to be spoken and then translated into source code as it happens to be with programming by voice. Hermans et al. [11] examined vocalization of code on group of novices: ten Dutch high school students that received twenty hours of Python lessons. In the end they did confirm that an issue exists, proposing that a standardize code phonology should be developed in the future. This could not only, as it is stated in the document, help to teach and learn programming, in pairwise programming, but speech recognition based development would benefit as well because each system of that kind could perform the same operations using the same spoken phrases.

Rodriguez-Cartagena et al. [15] also make this realization and propose a vocabulary and grammar to help in programming by voice in any C-based language. In order to evaluate, it they performed a study on programming community to receive feedback. The programming community consisted of random senior students and faculty members of the Computer Science and Computer Engineering Departments at the Polytechnic University of Puerto Rico and the University of Puerto Rico Río Piedras. This was supposed to be an initial phase for Kavita project, however at the time of writing this thesis the website of the project was unavailable.

2.12 Summary

CodeSpeech has certain commonalities with some of the previously mentioned projects, other things it does different, and certain good ideas could still be used in the future. It was decided for CodeSpeech to mainly perform AST operations rather than simply working on strings of text. This should give better control over the program structure e.g. when a whole block is to be copied or deleted, or a child node to be replaced without the need of text selection which in the case of voice control might be clumsy. From above mentioned projects only NaturalJava used the same approach. In the current state CodeSpeech consists of a built-in speech recognition system, while many other works such as VoiceCode or VocalProgrammer use external tools like e.g. Dragon NaturallySpeaking. In most cases these tools require an additional purchase from users. The initial idea of CodeSpeech was to be free of charge and therefore integrating an open-source SR tool seemed more viable. Additionally, there exists a possibility to adjust such a tool for higher performance in specialized tasks. The program might be, however, extended at some point to allow external software. CodeSpeech similarly to SPEED is integrated into Eclipse IDE and that makes it more than just a program to create code, but provides a way to expand and use all the functionality of a complete programming environment. Similarly to other works CodeSpeech uses grammar for commands. This might be a bit limiting solution, although it is the easiest to implement. An alternative would be to use a special language analysis tool like it was done in NaturalJava or SPEED. Maybe in the future such a software will replace grammar-based approach, but only if it turns out that it allows for control with more natural sentences. CodeSpeech could greatly benefit from better error correction with automatic ability to improve the results in a manner of VoiceCode. A very interesting idea to integrate usage of gestures was given in VGDE project and perhaps one day it will become a focus of CodeSpeech extension, possibly together with the idea of a different representation of the code (rather than standard text form). Many other aforementioned features have not been integrated due to time limitations but are considered for the future.

Chapter 3

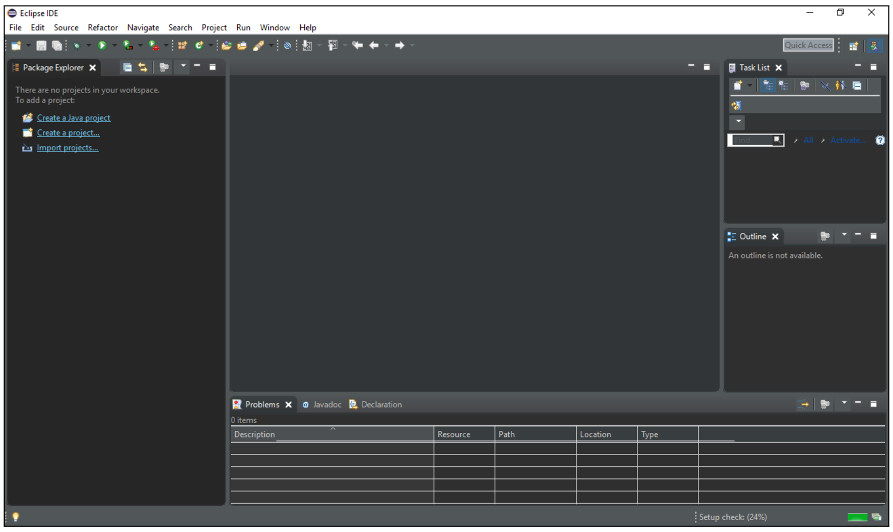
Tools

In order to create the CodeSpeech prototype, multiple different tools were combined together. Because this project was developed as an Eclipse plugin, IDE itself and Plug-In Development Environment (PDE) are used as a development platform. For speech recognition two different toolkits were tested, namely open-source CMUSphinx (in two versions - Pocketsphinx and Sphinx4) and a commercial Google Cloud Speech-to-Text. For analysis of recognized phrases against the grammar file ANTLR parser was utilized. This chapter will introduce and briefly describe each of the aforementioned tools.

3.1 Eclipse

Eclipse is free of charge, open-source Integrated Development Environment. Eclipse is mostly written in Java and primarily used for developing Java applications, although it also allows development in other programming languages such as languages from C family, Python, PHP, JavaScript etc. via plugins. The Eclipse Public License (EPL) is the fundamental license under which Eclipse projects are released. The initial launch was in November of 2001. It can be used at any of the most popular operating systems, such as Linux, Windows, Mac OSX, UNIX. The exemplary view of Eclipse IDE can be seen in Fig. 3.1.

It was selected as a development tool and the target platform for three main reasons among others. The first one being, the knowledge of and familiarity with the environment, which the author of this thesis gained during the many years of its use. The second is the fact that Eclipse is easily extendable via plugins (more on that later), therefore it seemed plausible to create an extension for it, especially because it provides a special set of tools that assist in the development of extension (described in the next section). Finally, when doing research about the potential development platforms it turned out that Eclipse JDT API contains all the necessary tools for Abstract Syntax Tree manipulation, which sounded like a perfect way for source code creation from within the program.



There are no projects in your workspace.
To add a project:

An outline is not available.

0 items

Figure 3.1: Eclipse IDE.

3.2 Eclipse PDE

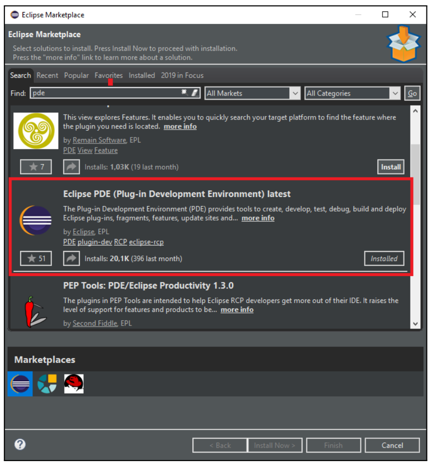
As mentioned before, Eclipse is mostly known as an IDE. According to Wayne Beaton of Eclipse Foundation [27], however, it was never intended to exclusively be one. It was developed to provide a platform unto which different tools could be integrated. Therefore in fact the core of Eclipse has small functionality by itself and the rest of the tools that are nowadays so broadly used are attached via plugins. That means, Eclipse is designed to be easily extendable. Because of its nature, Eclipse is a perfect tool to be used as a base for general purpose applications. These rich client applications, as well as, plugins for Eclipse core can be developed in Eclipse itself through the use of Plug-in Development Environment (PDE). Once PDE is installed it is enables creation of a new plugin project. The project requires a proper set up at first. Eclipse plugins are based on Open Service Gateway Initiative (OSGi) bundles [31]. OSGi is used to manage them in Eclipse application. A plugin must contain a manifest file (*MANIFEST.MF*) with valid OSGi headers for name and version (similarly to Maven). Manifest file is extremely important. It describes the content of a plugin necessary for Eclipse during run-time, such as dependencies, classpath, what is exported etc. Information about the build is held in *build.properties* file. Modular nature of plugins allow them to use other decoupled elements and to be used by other elements as well. This is done by so called extensions and extension points. An extension is e.g. a UI element that runs a specific command (which is an extension by itself). CodeSpeech has one UI extension added to the menu - a toggle button that switches the listening on or off. These are managed by another file, namely *plugin.xml*. PDE provides a full featured editor for modifying all three files: *MANIFEST.MF*, *build.properties* and *plugin.xml*.

Once everything is set up, development is focused around a class called the `Activator`. It is the starting point of any plugin. `Activator` class extends the `Plugin` and it is responsible for other classes. The class is initialized whenever any member of the plugin is referenced.

To familiarize the reader with the plugin development using PDE a small example is presented below. The example is explained in detail through the description of the process together with screenshots. The plugin consists of a single toolbar button which displays a message after being clicked.

3.2.1 Necessary software

As it was mentioned before, the first thing needed is an instance of Eclipse IDE installed in the computer. Eclipse can be downloaded from the official website [1]. After that, an enhancement of PDE has to be integrated into the IDE. This can be done via Eclipse Marketplace, accessible from within Eclipse itself, through its menu (Help -> Eclipse Marketplace). In addition to that, yet another software has to be installed, namely Eclipse RCP Target Components. This can be done also through IDE's menu (Help -> Install New Software...). Fig. 3.2 show how the corresponding download windows look like. After all of these steps are done, the environment is ready to create a plugin.


Eclipse Marketplace

Eclipse Marketplace

Select solutions to install. Press Install Now to proceed with installation.
Press the "more info" link to learn more about a solution.

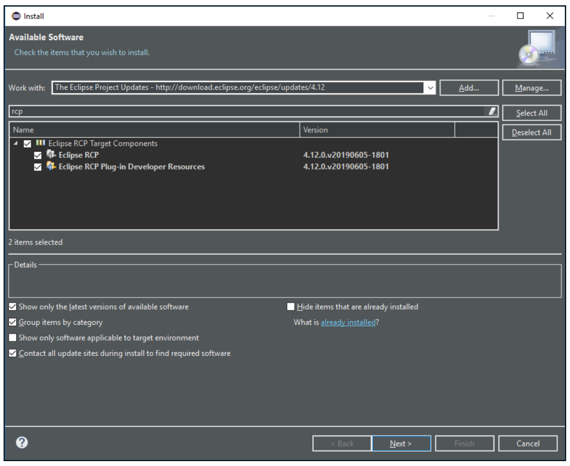
Find: pde — All Markets — All Categories

This view explores Features. It enables you to quickly search your target platform to find the feature where the plugin you need is located. more info
by Remain Software, EPL
PDE View Feature
★ 7 — Installs: 1,03K (19 last month) — Install

Eclipse PDE (Plug-in Development Environment) latest
The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and... more info
by Eclipse, EPL
PDE plugin-dev RCP eclipse-rcp
★ 51 — Installs: 20,1K (396 last month) — Installed

PEP Tools: PDE/Eclipse Productivity 1.3.0
The plugins in PEP Tools are intended to help Eclipse RCP developers get more out of their IDE. It raises the level of support for features and products to be... more info
by Second Fiddle, EPL

(a)


Install

Available Software

Check the items that you wish to install.

Work with: The Eclipse Project Updates - http://download.eclipse.org/eclipse/updates/4.12

rcp

Name	Version
☑ Eclipse RCP Target Components	
☑ Eclipse RCP	4.12.0.v20190605-1801
☑ Eclipse RCP Plug-in Developer Resources	4.12.0.v20190605-1801

2 items selected

Details

☑ Show only the latest versions of available software
☑ Group items by category
☐ Show only software applicable to target environment
☑ Contact all update sites during install to find required software
☐ Hide items that are already installed
What is already installed?

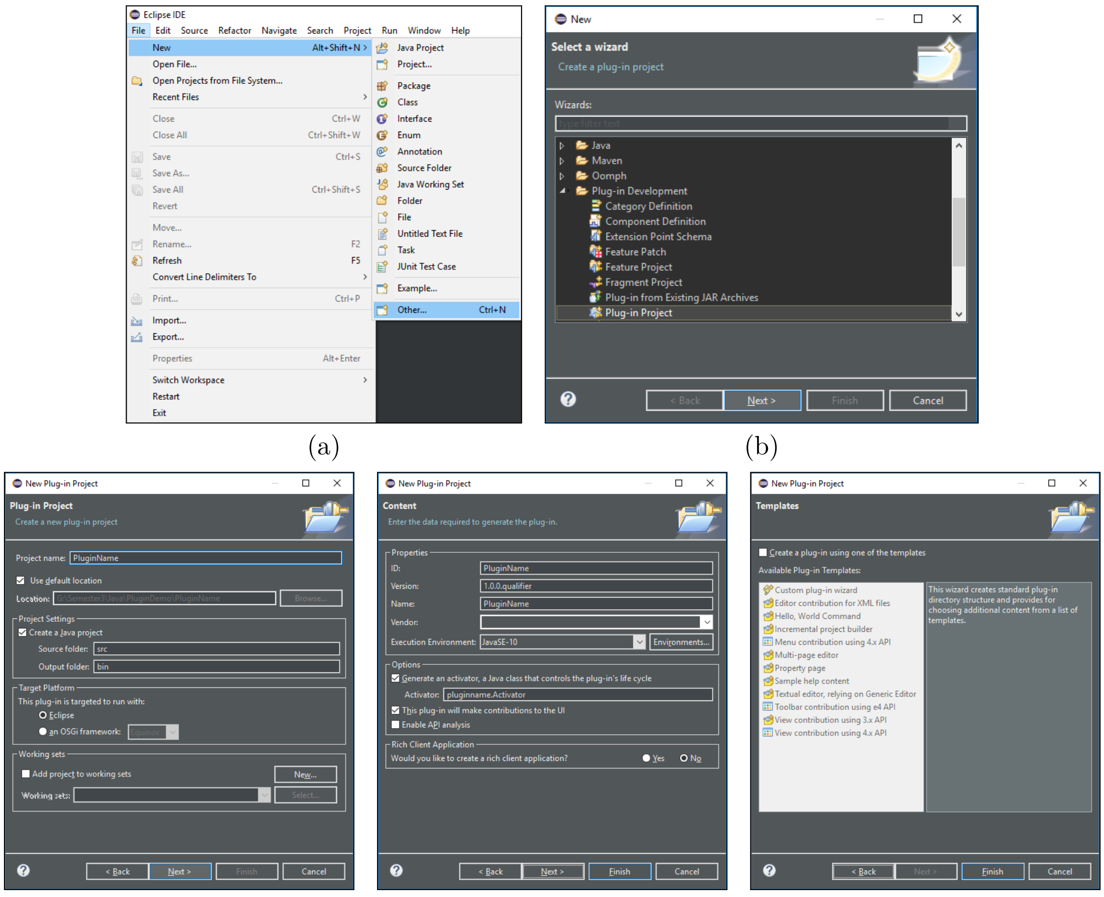
(b)

Figure 3.2: Eclipse Marketplace window for PDE donwload (a) and software installation window with Eclipse RCP Target Components Software (b)

3.2.2 Project creation

A plugin can be created in a similar manner as any other type of a project. A creation wizard leads the user through the setup process. The wizard can be opened using menu (File -> New -> Other...). In the pop up window a *Plug-in project* can be selected under the *Plug-in Development* directory. The next consecutive wizard's windows allow for setting up things like plugin's name, id, version as well as setting an `Activator` and a couple of other options. The last page gives an opportunity to select one of a few plugin templates which comes with some predefined structures. In this example no template is used. Fig. 3.3 shows the sequence of previously mentioned steps.

[1]https://www.eclipse.org/eclipseide/



(c)
Project name: PluginName
[✓] Use default location
Location: G:\Semester3\Java\PluginDemo\PluginName
Project Settings
[✓] Create a Java project
Source folder: src
Output folder: bin
Target Platform
This plug-in is targeted to run with:
[✓ Eclipse] an OSGi framework: Equinox
Working sets
[ ] Add project to working sets
Working sets:

(d)
Properties
ID: PluginName
Version: 1.0.0.qualifier
Name: PluginName
Vendor:
Execution Environment: JavaSE-10
Options
[✓] Generate an activator, a Java class that controls the plug-in's life cycle
Activator: pluginname.Activator
[✓] This plug-in will make contributions to the UI
[ ] Enable API analysis
Rich Client Application
Would you like to create a rich client application? Yes [✓ No]

(e)
[ ] Create a plug-in using one of the templates
Available Plug-in Templates:
Custom plug-in wizard
Editor contribution for XML files
Hello, World Command
Incremental project builder
Menu contribution using 4.x API
Multi-page editor
Property page
Sample help content
Textual editor, relying on Generic Editor
Toolbar contribution using e4 API
View contribution using 3.x API
View contribution using 4.x API

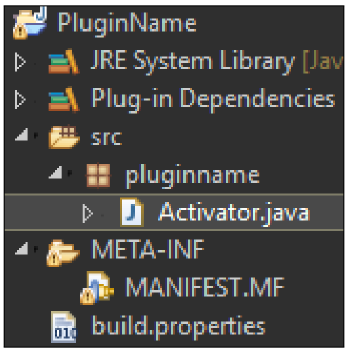
This wizard creates standard plug-in directory structure and provides for choosing additional content from a list of templates.

Figure 3.3: Consecutive steps of plugin project creation. Launching creation wizard (a & b), project options wizard (c), plugin options wizard (d) and template wizard (e).

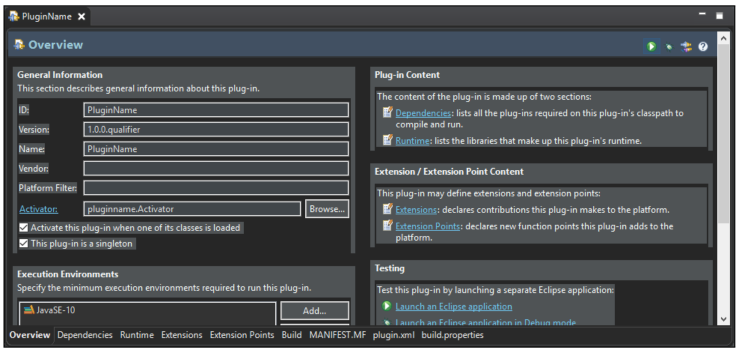
Once the project is created it can be seen in *Package Explorer*. It already contains *build.properties* and *MANIFEST.MF* files and `Activator` class. By clicking on the manifest file a special editor opens up on *Overview* tab. Fig. 3.4 presents generated project structure and depicts manifest editor.

3.2.3 Adding extensions

Adding a toolbar button requires extending the *org.eclipse.ui.menus* extension point. This can be in the *Extensions* tab, where by clicking the *Add* button another wizard appears. This wizard allows us to browse, search and select necessary extension point. When extension point of interest is found and confirmed with *Finish* button, the new extension point is visible as added into the list. By right-clicking the item a drop down menu is displayed, opening certain options for this item. To add a toolbar, a new menu contribution needs to be created first. Its *locationURI* field must be set to *toolbar:org.eclipse.ui.main.toolbar*. The location specifies where the extension will be placed. The contribution has to be extended by a toolbar with specified id. In order



(a)


PluginName

Overview

General Information
This section describes general information about this plug-in.
ID: PluginName
Version: 1.0.0.qualifier
Name: PluginName
Vendor:
Platform Filter:
Activator: pluginname.Activator Browse...
☑ Activate this plug-in when one of its classes is loaded
☑ This plug-in is a singleton

Execution Environments
Specify the minimum execution environments required to run this plug-in.
JavaSE-10 Add...

Plug-in Content
The content of the plug-in is made up of two sections:
Dependencies: lists all the plug-ins required on this plug-in's classpath to compile and run.
Runtime: lists the libraries that make up this plug-in's runtime.

Extension / Extension Point Content
This plug-in may define extensions and extension points:
Extensions: declares contributions this plug-in makes to the platform.
Extension Points: declares new function points this plug-in adds to the platform.

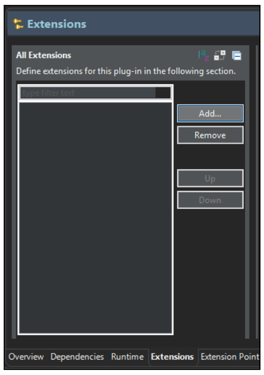
Testing
Test this plug-in by launching a separate Eclipse application:
Launch an Eclipse application
Launch an Eclipse application in Debug mode

Overview Dependencies Runtime Extensions Extension Points Build MANIFEST.MF plugin.xml build.properties

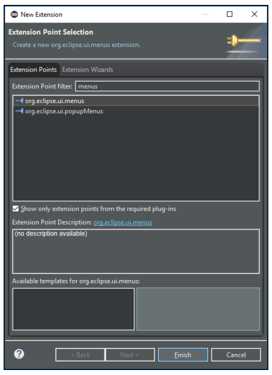
(b)

Figure 3.4: Project structure(a), manifest editor (b).

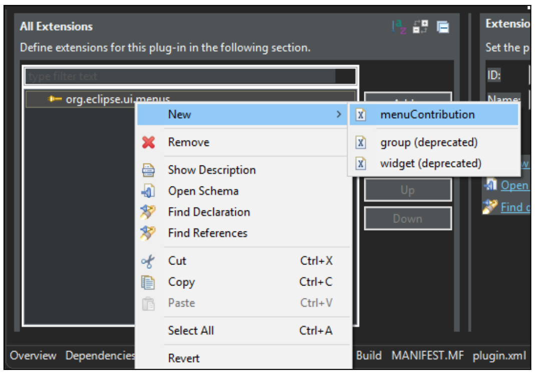
to provide a functionality to it, a command has to be created first. Analogically, at the beginning *org.eclipse.ui.commands* extension point has to be added to the list, then extended by a new command, whose id and name should be specified. Each of the steps is presented in Fig. 3.5.


Extensions
All Extensions
Define extensions for this plug-in in the following section.
Add...
Remove
Up
Down
Overview Dependencies Runtime Extensions Extension Point

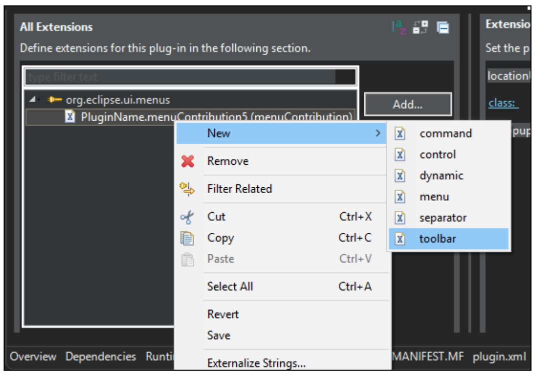
(a)


New Extension
Extension Point Selection
Create a new org.eclipse.ui.menus extension.
Extension Points Extension Wizards
Extension Point filter: menus
org.eclipse.ui.menus
org.eclipse.ui.popupMenus
☑ Show only extension points from the required plug-ins
Extension Point Description: org.eclipse.ui.menus
(no description available)
Available templates for org.eclipse.ui.menus:
< Back Next > Finish Cancel

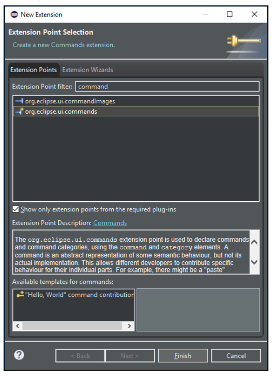
(b)



(c)



(d)


New Extension
Extension Point Selection
Create a new Commands extension.
Extension Points Extension Wizards
Extension Point filter: command
org.eclipse.ui.commandImages
org.eclipse.ui.commands
☑ Show only extension points from the required plug-ins
Extension Point Description: Commands
The `org.eclipse.ui.commands` extension point is used to declare commands and command categories, using the `command` and `category` elements. A command is an abstract representation of some semantic behaviour, but not its actual implementation. This allows different developers to contribute specific behaviour for their individual parts. For example, there might be a "paste"
Available templates for commands:
"Hello, World" command contribution
< Back Next > Finish Cancel

(e)

Figure 3.5: Extenstion points list in Extenstions tab (a), addition of *org.eclipse.ui.menus* (b), addition of menu contribution and a toolbar (c & d), addition of *org.eclise.ui.commands* (e).

When a command is triggered, its handler performs specified action. To assign a handler to the command, its name needs to be put into the *defaultHandler* field (or selected via *Browse* button to the right). In this case a new handler is to be implemented in a form of a new class that extends `AbstractHandler`. Resulting class presented in Program 3.1 will only display a dialog with text. Now that the command is created it can be used in a toolbar button. Assigning the command to the toolbar is done in a similar manner as the rest of the extensions. It is important to give correct command's id in a field named *commandId*. These steps are depicted in Fig. 3.6. An image can be added to the button via *icon* field, this time it is 16x16 pixels *.png* file. If everything was done correctly, the program can be run as "Eclipse Application". The plugin's toolbar should be visible in a menu with the chosen image and once clicked, the dialog should appear as shown in Fig. 3.7. That concludes the example. More detailed information can be found in the official documentation [29].

Program 3.1: Start of interpretation procedure.

```
public class ToolbarClickHandler extends AbstractHandler {
  @Override
  public Object execute(ExecutionEvent event) throws ExecutionException {
    MessageDialog.openInformation(HandlerUtil.getActiveWorkbenchWindow(
                event).getShell(), "Info", "You clicked the button!");
    return null;
  }
}
```

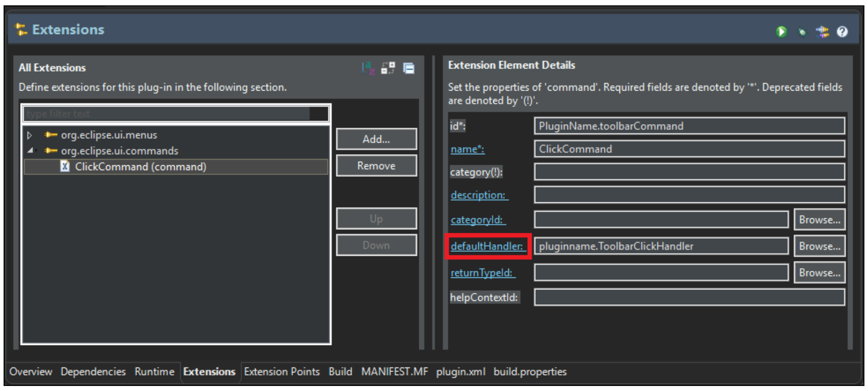
3.3 Speech Recognition tools

Three different speech recognition tools are integrated into the project - two are different implementations of CMUSphinx toolkit, which is open-source and free of charge, and one is using Google Cloud Speech-to-Text API. Subsections below describe each of them in more detail. Before that, however, at the beginning a short introduction into how speech recognition works is given.

3.3.1 Explaining speech recognition

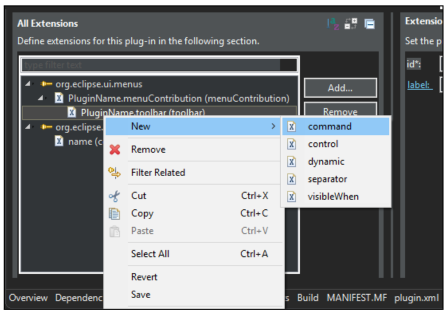
In a nutshell, humans communicate with each other using language. This languages consists of sentences, and these consists of words. Focusing only on the spoken aspect of a language, words are made out of subwords such as syllables, which in turn are made out of units of sound called phonemes. It is a simplification, however, as the sound of the phonemes can vary depending on the speed of speech, context, speaker, accent etc. In English language there are roughly 40 phonemes. It is these elements that the computer tries to match in order to recognize a word.

At first, a couple of sentences on how voice is being transferred to the computer. Voice is a sound wave created by a vocal apparatus that produces vibrations in the air. A microphone converts the sound into electrical signal. This signal (as the sound itself) is continuous (analog). In order for a computer to understand this signal it has

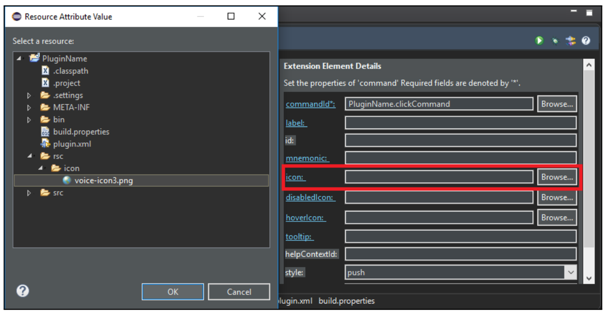

Extensions
All Extensions
Define extensions for this plug-in in the following section.
org.eclipse.ui.menus
org.eclipse.ui.commands
ClickCommand (command)
Add...
Remove
Up
Down
Extension Element Details
Set the properties of 'command'. Required fields are denoted by '*'. Deprecated fields are denoted by '(!)'.
Browse...
Overview Dependencies Runtime Extensions Extension Points Build MANIFEST.MF plugin.xml build.properties

id*: PluginName.toolbarCommand
name*: ClickCommand
category(!):
description:
categoryId:
defaultHandler: pluginname.ToolbarClickHandler
returnTypeId:
helpContextId:

(a)


All Extensions
Define extensions for this plug-in in the following section.
org.eclipse.ui.menus
PluginName.menuContribution (menuContribution)
PluginName.toolbar (toolbar)
New
Remove
Filter Related
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Select All Ctrl+A
Revert
Save
command
control
dynamic
separator
visibleWhen
Add...
Remove
Overview Dependencies Build MANIFEST.MF plugin.xml

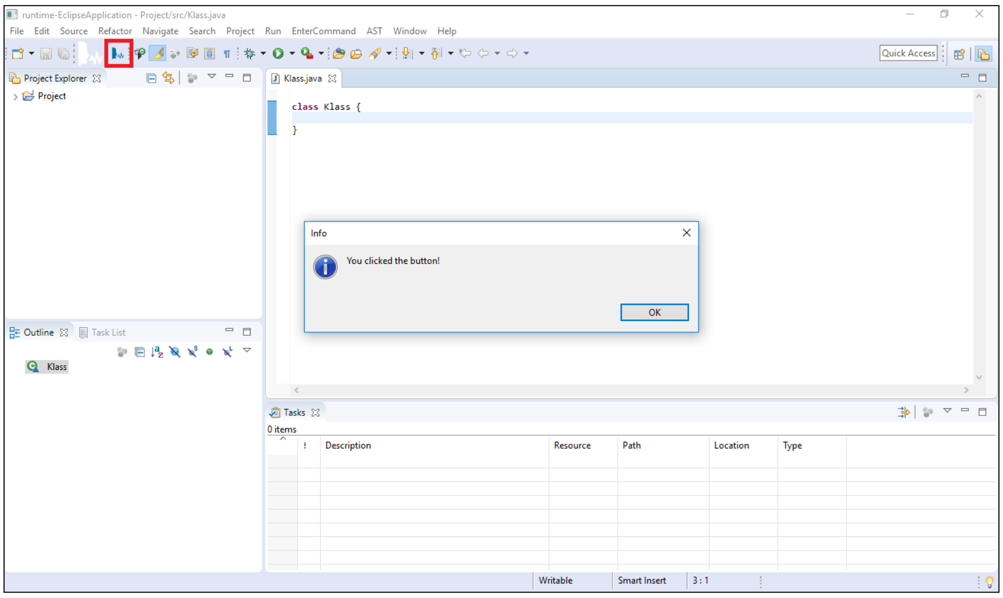
(b)



commandId*: PluginName.clickCommand
label:
id:
mnemonic:
icon:
disabledIcon:
hoverIcon:
tooltip:
helpContextId:
style: push

(c)

Figure 3.6: Field for command's default handler (a). Addition of command to the toolbar (b & c).


runtime-EclipseApplication - Project/src/Klass.java
File Edit Source Refactor Navigate Search Project Run EnterCommand AST Window Help
Quick Access
Project Explorer
Project
Klass.java
Outline
Task List
Klass
Tasks
0 items
Description Resource Path Location Type
Writable Smart Insert 3 : 1

```
class Klass {
}
```

Info

You clicked the button!

OK

Figure 3.7: Resulting plugin.

to be converted into a digital one. This is done by analog-to-digital converter (ADC), which performs certain operations such as sampling, filtering, normalization, adjusting to match the speed of a template etc. Once the signal is ready, it is then divided into small portions (frames) of a few hundredths or thousandths of seconds which are a subset of phonemes called senones. These segments are used to match known phonemes.

The rest of the process is belongs to Machine Learning domain. For each frame a number of features are extracted. Each feature is a characteristic aspect of the segment which together with others creates a feature vector - a fingerprint of a kind. These features are used to match a model using complex mathematical functions and probability to determine the most likely outcome. Because of computational limitations and complexity of feature matching, the number of features should be as small as possible, yet the features should be strong enough to enable distinction between the segments.
In speech recognition the most commonly used model is called Hidden Markov Model (HMM). During the process of matching each phoneme is assigned a specific probability score based on previously done training and a dictionary. The process does not consider one phoneme at a time, but it also takes into account neighbouring phonemes, therefore it is contextual. It is done on the level of building words, but also the whole phrases.

There are two models that are necessary for speech recognition: acoustic and language model. The former contains acoustic properties for each senone. This model can be context-dependent or context-independent. The latter is used to restrict the search for words by defining which word could follow the one preceding it. In result this significantly quickens the process by focusing on the most probable words. Otherwise, having a vocabulary of 60.000 words and a sequence of three words the program would need to check 216 trillion possibilities.
A component that is required to combine these two models provides some sort of a mapping between the phonemes and words. This can either be done by so called phonetic dictionary, which is a list of words with phonemes representing them, or by more efficient complex mathematical function achieved by machine learning algorithm.

3.3.2 CMUSphinx

The initial idea was to base the whole project on available free of charge software elements. An open-source Speech Recognition engine was therefore preferable. After a small research CMUSphinx was selected as it was recommended most often. It was said that it is accurate enough, easy to implement and with very broad documentation. Its documentation is indeed quite well described and is very helpful [28]. It is stated there that CMUSphinx is *leading speech recognition toolkit with various tools used to build speech applications.* On top of that *CMUSphinx contains a number of packages for different tasks and applications.* The toolkit provides different tools, including two recognizer libraries: Pocketsphinx and Sphinx4. Pocketsphinx is a lightweight library written in C, recommended when speed and portability are the main factors or when recognition is to be run on embedded device. Sphinx4 on the other hand is easily adjustable, modifiable recognizer written in Java. It is said that its flexibility makes it possible to quickly build a system. When it comes to selection between the two, it is not accuracy that should be the main criteria but rather the nature of application. In this project Sphinx4 was the first choice mainly for the sole reason, that it is a pure

Java library and it is simple to integrate it to Java project, which CodeSpeech is. After further reading, however, the author of this thesis decided to also give Pocketsphinx a try. That is because of the additional recognition method that Pocketsphinx provides which Sphinx4 does not, namely keyphrase recognition. Other two methods that are available for both technologies are continuous speech and grammar based recognition. All three of the previously mentioned methods of recognition will be explained later.

Sphinx4

As stated in the documentation, integration of Sphinx4 was relatively easy and quick. The only thing that was needed to use its API were two Java library files, namely sphinx4-data.jar and sphinx4-core.jar. The former consists of acoustic model, dictionary and language model, the latter is API itself. After giving it a try it turned out that recognition of continuous speech did not work well for the author of the thesis. The accuracy, even though not measured, was clearly poor.

PocketSphinx

Integrating PocketSphinx into the project was a completely different story. It lasted for plenty of hours that were spent on solving continuously coming up issues occurring on different stages. Problems followed the whole process: from correctly building libraries from the source, through usage of Swig to build JNI Java files from C code (although bless the creators of CMUSphinx that this possibility is present and the whole library does not have to be translated to JNI manually) to finally use the generated JNI files as a library in a project. Error messages were often misleading and resulted in additional hours spent on looking for a mistake (which are very easy to make). To be fair, PocketSphinx was never intended to be used in Java projects (that is what Sphinx4 is for), and yet a possibility exists.

When finally all of the necessary libraries were built and loaded, the implementation could be continued. At first an Android project using PocketSphinx available online was used as a template. The accuracy of PocketSphinx was similarly poor to Sphinx4's, nevertheless the results were being returned faster.

With both technologies giving not satisfying enough results, the author decided to try to adapt models used by CMUSphinx in order to improve the accuracy of the recognition.

Model adaptation

In cases of low accuracy the documentation of CMUSphinx proposes to adapt acoustic model. This can be done relatively easily using Sphinxbase library which *provides common functionality across all CMUSphinx projects*. Sphinxbase is provided together with PocketSphinx package. Adaptation was performed based on corresponding chapter of the documentation. Accordingly, twenty audio transcriptions consisting of the author reading multiple words were recorded, proper text files were created and finally the model was adopted. It did not bring the results that were hoped for, unfortunately, accuracy remained poor. It is possible, that for a significant improvement even more recordings (at least hundred) are necessary, because of time limitation however, this process was skipped.

Since the adaptation of the acoustic model did not end in significant increase of accuracy of recognition, another means of improvement mentioned in the documentation were performed. These included building a new language model, new dictionary and combination of all three steps. These tasks were quite difficult to do on Windows OS and turned out to be easier on Linux. In the end no success was achieved.

The creators of CMUSphinx explained that the low accuracy might be related to the author's accent. The models were built for American speakers and thus does not work well for people talking differently. Even British people won't be able to achieve high accuracy unless they build a new model their own voices. Building the new model from scratch requires a lot of time, plenty of hours of recordings and is a very complicated process. Due to the limited amount of time it was decided to move on and find another speech recognition tool.

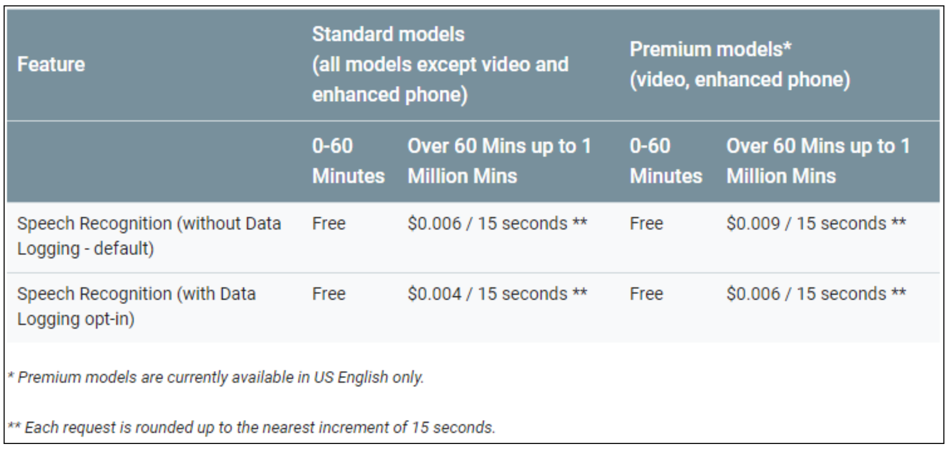
3.3.3 Google Cloud Speech-to-Text

Google Cloud Speech-to-Text is a service that enables developers to convert audio to text by applying powerful neural network models hosted in a cloud. It recognizes 120 languages and variants, and has an API that is easy to use. On the official website of the service it is possible to try it out before making a decision, and without a doubt this recognition is far better than the one provided by CMUSphinx. API was indeed very simple to integrate to Java project and worked almost out of the bat. This service is not free of charge however. The pricing as it was at the time of writing this thesis is presented in Fig. 3.8. As can be seen, there exists a free tier up to 60 minutes of recording that lasts for a year. This is very fortunate, because the initial assumption was to create programming by voice prototype without any money spend. The free tier allows the author to create the prototype and see if the idea of programming by voice in a plugin is possible at all. Once the hypothesis is confirmed, more time can be put into either enhancing current CMUSphinx models, creating a completely new one or applying yet another free-of-charge tool with good enough accuracy.

Google Speech-to-Text being a cloud service, obviously requires Internet access. The recording of voice is done on the local machine, but once an utterance is finished, it is being sent over to Google server and there neural network models perform recognition. After it is done the result is sent back via callback. That is another difference with previously mentioned technologies, which run entirely on local machine and do not require an Internet connection.

3.3.4 Comparison of SR tools

A summary of certain features belonging to aforementioned tools is presented in the Tab. 3.1. This table might help in making a decision as to which technology should be used in the development of future Java projects. The difficulty of integration is of course relative, however, the fact that Sphinx4 and Google Cloud Speech-to-Text tools already have existing, ready to use APIs written in Java while PocketSphinx does not, plays a big part here. PocketSphinx library has to be built, then JNI code generated etc. The time needed for for all theses steps might differ depending on the programmers skills



Feature	Standard models (all models except video and enhanced phone)		Premium models* (video, enhanced phone)	
	0-60 Minutes	Over 60 Mins up to 1 Million Mins	0-60 Minutes	Over 60 Mins up to 1 Million Mins
Speech Recognition (without Data Logging - default)	Free	$0.006 / 15 seconds **	Free	$0.009 / 15 seconds **
Speech Recognition (with Data Logging opt-in)	Free	$0.004 / 15 seconds **	Free	$0.006 / 15 seconds **

** Premium models are currently available in US English only.*

*** Each request is rounded up to the nearest increment of 15 seconds.*

Figure 3.8: Pricing of Google Cloud Speech-to-Text service (taken from official website [30]).

and knowledge, nevertheless the number of steps required for integration is significantly higher in comparison to the other two. It is of course understandable as PocketSphinx was never intended to be used for Java projects.
In addition, all engines were tested in terms of recognition time and Word Error Rate (WER) metric. The formula for WER is

$$WER = \frac{I + D + S}{N},$$

where I is the number of inserted words, D is the number of deleted words, S is the number of substituted words and N is the number of words in the original (reference) sentence. In controversial situations like e.g. when one word from the reference text was recognized as n words, one substitution and n - 1 insertions were counted. In a reverse situation, when multiple words were recognized as one, the entry had one substitution and n - 1 deletions. In order for the comparison to be more accurate, test was performed on recordings rather than live speech. The reason for that is to ensure that exactly the same audio is analysed by each software. For this purpose ten sentences were recorded and used by each tool's file analyser. Resulting sentences were written to the file and then WER was manually calculated.

The results of this test show that when it comes to performance Google Cloud Speech-to-Text is clearly the winner. Not only it does the recognition quickly despite the need of sending and receiving data through the Internet, but also has the smallest WER score of 17%. PocketSphinx recognizes in a similar speed (in this experiment slightly faster), but its WER is very high (around 95%). Sphinx4 did a bit better than PocketSphinx in terms of recognition, but was incredibly slow (almost 10 times slower than the other two). Taking all that into consideration, Google Cloud Speech-to-Text was used as a main speech recognition engine, although all three were integrated into the system. Perhaps in the future CMUSphinx models will be trained to increase the performance.

Table 3.1: Comparison of used SR technologies in regard of features important for Java developers.

	Sphinx4	*PocketSphinx*	*Google Speech-to-Text*
Integration difficulty	Easy	Difficult	Easy
Price	Free	Free	Free-tier, paid
Internet connection	No	No	Required
WER [%]	88,2	94,3	17
Average time [s]	19,76	2,12	2,17

3.4 ANTLR

The end goal of the finished programming by voice product is to perform operations using natural sentences. At the end of the previous chapter an issue related to the way code is read by different people is discussed. The same problem might appear when it comes to steering the program by voice through the use of different commands. Taking that into consideration, it was decided that for the control over the program by voice commands an interpretation of specified grammar will be used. Although grammars are usually quite strict when it comes to token recognition because each of the required elements must occur in a right position, they also allow a certain amount of flexibility by enabling representation of tokens by different keywords/phrases. Once decided upon this approach, a proper tool had to be selected for parsing and managing grammars and finally the choice was ANTLR (ANother Tool for Language Recognition). ANTLR as described on the official website

> is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees [25].

ANTLR turned out to be exactly what was needed, giving a lot of options in the way the sentence is parsed. This flexibility is a great advantage of ANTLR. Below more information about working with ANTLR are given.

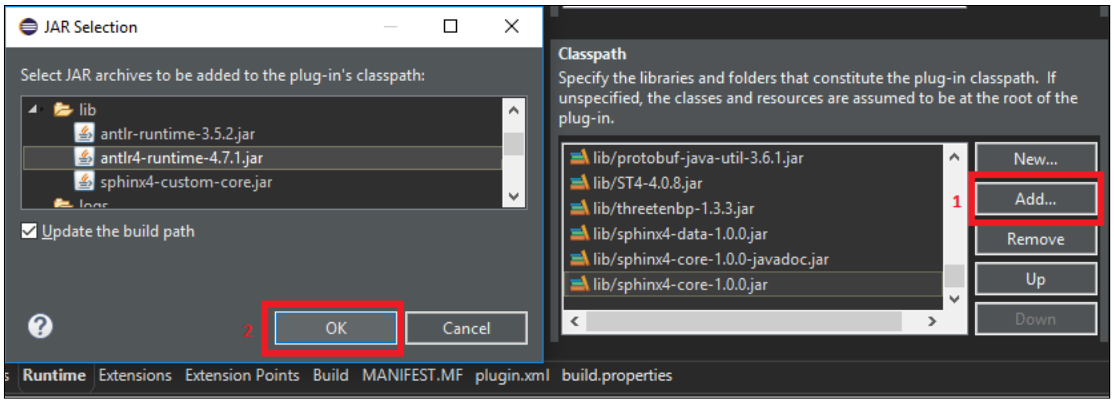
3.4.1 Typical workflow

At first ANTLR tools have to be installed and then grammar file with *.g4* extension created. Then, running ANTLR tools against the grammar generates code that enables handling each specific token or keyword at any time. Whenever the change in grammar is made, new files have to be generated and applied to the project. The most important from the point of view of developer are listeners or visitors files (depending on the option). They provide methods that are triggered whenever a token is entered or exited. Many custom listeners or visitors can be implemented, each allowing for alternative interpretation of the same token depending on a current state of the program. The whole process was divided into three parts: installation of ANTLR tools, writing a grammar and generation of the files.

ANTLR tools

ANTLR consists of two parts: the tool used to generate lexer and parser, and the runtime needed to run them by the end software. The tool is the Java program and it requires at least Java 1.7 installed on a working machine. To install the program it needs to be downloaded from the official website from a download section and then added to the classpath of the system. When done, `antlr4 <options> <grammar-file-g4>` command can be used in order to generate files needed by the end software. Options allow generation thereof to a specific language other than Java (which is set as default) e.g. Python or JavaScript, or to choose between generation of listeners (default) or visitors.
This is one way of using ANTLR. During this project yet another approach was used. Generation of needed files based on the grammar was done through another external ANTLR project. It is a simple Java project which uses ANTLR libraries via Maven. The grammar is placed in project's *src/main/antlr4* folder. Generation is then done using `mvn package` command.

The end Java program that will use ANTLR can be set up either by Gradle, Maven or manually. Because of the conflict that occurs between Eclipse plugin project and other manifest-based ones, for now the CodeSpeech is set up manually. Addition of external libraries such as antlr4-runtime.jar to Eclipse plugin project can be done via *Runtime* tab of the manifest editor in the *Classpath* area as shown in Fig. 3.9.


JAR Selection

Select JAR archives to be added to the plug-in's classpath:

- lib
  - antlr-runtime-3.5.2.jar
  - antlr4-runtime-4.7.1.jar
  - sphinx4-custom-core.jar

☑ Update the build path

Classpath

Specify the libraries and folders that constitute the plug-in classpath. If unspecified, the classes and resources are assumed to be at the root of the plug-in.

- lib/protobuf-java-util-3.6.1.jar
- lib/ST4-4.0.8.jar
- lib/threetenbp-1.3.3.jar
- lib/sphinx4-data-1.0.0.jar
- lib/sphinx4-core-1.0.0-javadoc.jar
- lib/sphinx4-core-1.0.0.jar

Figure 3.9: Addition of external library.

Grammar

Grammar consists of lexer and parser rules. Lexer, also known as tokenizer analyzes text and assigns a proper token to parts of it. Parser takes these tokens and organizes them into abstract syntax tree. Both parser and lexer perform their tasks using so called rules. In ANTLR lexer and parser rules are distinguished by the first character of the rule's name: parser rules start with lower case and lexer rules with upper case. For better readability it is common to write the name of the lexer rules completely in upper case. Lexer rules are usually put at the bottom of the grammar, however, from the ANTLR's point of view it does not matter, it is just a standard of organization. It is possible to divide parser and lexer rules to two separate files. In Program 3.2 both types of rules are in one file, with lexer rules being at the bottom.

Program 3.2: Example of grammar file's content. This grammar allows creation of simple mathematical equations.

```
grammar BasicMath;

/*
 * Parser Rules
 */
program:  (equation NEWLINE)* ;
equation: equation ('*'|'/') equation
    | equation ('+'|'-') equation
    | INT
    | '(' equation ')'
    ;

/*
 * Lexer Rules
 */
fragment DIGIT : [0-9] ;
INT     : DIGIT+ ;
NEWLINE : [\r\n]+ ;
WHITESPACE : (' ' | '\t') ;
```

As it can be seen, the content of a rule is placed between a colon and a semi-colon defined after the name. A `fragment` rule allows to create reusable blocks for lexer rules. Parser rule can consist of parser and lexer rules. It can also be seen that ANTLR allows recursion, so referencing itself in the rule. A few symbols that exist in this example are: "|" for an alternative, "+" for one or more occurrence, "*" for zero or more, parentheses for definition of a subrule. Characters, whole words or phrases that are to be recognized are defined in single quotes. More examples, symbols and other interesting information can be found in the official documentation [26].

Generation

Once the grammar is finished a time has come for generation. Without regard for the method of performing that procedure (two were mentioned before), the generated files have to be moved to the project source folder in order to be used. Every time the grammar is modified, the process has to be repeated to apply the changes. This occurs quite often during the development and might be bothersome, however, a simple shell script can help in automation of the whole procedure, which in the end can save a lot of time. The script at first initializes generation by running `mvn package` in the home folder of ANTLR project. Then it adds proper package name via combination of `echo`, `cat` and `mv` commands and later moves them, to the target directory via `mv`. This solution speeds up the process, leaving refreshing the project as the only thing to do.

The generated files are named according to the grammar's file name. If the grammar is named *BasicMath*, then the resulting files will be *BasicMathLexer.java*, *BasicMath-Parser.java*, *BasicMathListener.java* and *BasicMathBaseListener.java.* File extension of course depends on the selected target programming language. In case of choosing visitors to be created those would replace listener files. In addition, *.interp* and *.tokens* files appear, but those are not needed by the runtime. Program 3.3 presents an interface contained in *BasicMathListener.java* file. Its implementation in form of abstract class is consisted in *BasicMathBaseListener.java.*

Program 3.3: A generated listener.

```java
// Generated from BasicMath.g4 by ANTLR 4.7.1
import org.antlr.v4.runtime.tree.ParseTreeListener;

/**
 * This interface defines a complete listener for a parse tree produced by
 * {@link BasicMathParser}.
 */
public interface BasicMathListener extends ParseTreeListener {
  /**
   * Enter a parse tree produced by {@link BasicMathParser#program}.
   * @param ctx the parse tree
   */
  void enterProgram(BasicMathParser.ProgramContext ctx);
  /**
   * Exit a parse tree produced by {@link BasicMathParser#program}.
   * @param ctx the parse tree
   */
  void exitProgram(BasicMathParser.ProgramContext ctx);
  /**
   * Enter a parse tree produced by {@link BasicMathParser#equation}.
   * @param ctx the parse tree
   */
  void enterEquation(BasicMathParser.EquationContext ctx);
  /**
   * Exit a parse tree produced by {@link BasicMathParser#equation}.
   * @param ctx the parse tree
   */
  void exitEquation(BasicMathParser.EquationContext ctx);
}
```

As it can be seen, for each parser rule there exist two methods: for entering and exiting specified parser rule. This allows the end program for performing specific actions depending on the implementation. Because many custom listeners can be created and changed back and forth during runtime, a different operation can be performed on the same keyword depending on the context. This elasticity of ANTLR, as well as ease of use, well written and broad documentation and active support were the main factors that decided upon selecting ANTLR as an interpretation tool.

3.4.2 ANLTR plugin

To make the work with ANTLR easier, it can also be integrated into many IDEs. There exists a plugin for Eclipse called *AntlrDT Tools Suite for Eclipse*, which provides following features:

- Advanced Syntax Highlighting
- Automatic Code Generation (on save)
- Manual Code Generation (through External Tools menu)
- Code Formatter (Ctrl+Shift+F)
- Syntax Diagrams
- Advanced Rule Navigation between files (F3 or Ctrl+Click over a rule)
- Quick fixes

Syntax diagram is a very helpful feature which helps to visualise how the grammar is constructed. An example can be seen in Fig. 3.10.

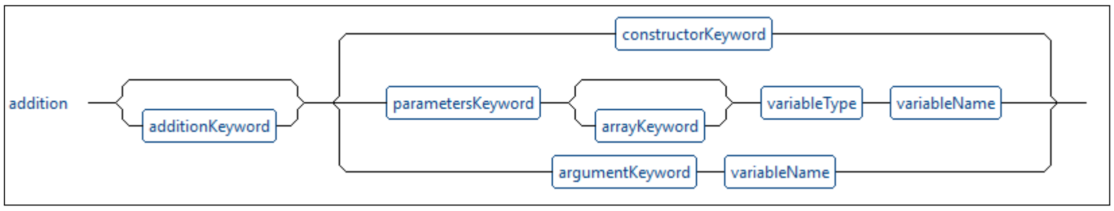


Figure 3.10: An example of syntax diagram.

Chapter 4

Method

This chapter goes through requirements definition and design stages. It is here, where one can find specification of the functional and non-functional requirements, use cases and a plan of general architecture of the system.

4.1 Requirements

This section is devoted to stating requirements. Functional and non-functional requirements have been divided into two subsections. At the end a couple of constraints that the project has are mentioned.

4.1.1 Functional requirements

Functional requirements should define what the system ought to do. Below the main requirements are presented in the form of a list.

The product shall:

- properly recognize speech,
- create source code,
- modify source code,
- run a program,
- debug a program,
- allow navigation in the file,
- allow navigation in the workspace.

The system's main functionality can be stated as "translating user's speech into an operation that is to be performed inside IDE". Each particular requirement corresponds to one type of operation. In other words, the system should respond to user's input, which is always audio data. The resulting output is an action performed by the system. When the user wants, for example, to create a for loop, he/she has to speak out a proper sentence which the system should interpret.

Because CodeSpeech is to allow writing programs by voice, of course it needs to be able to perform every task that is required in programming. It needs to provide a way to create source code, modify thereof, navigate inside files as well as in between files.

Running and debugging the program are also inherent parts of programming, therefore should also be possible.

These requirements are a certain simplification, as e.g. the term “create source code” could mean creation of variables, methods or other structures, similarly “modify source code” could mean deletion of a line, changing the name of the structure, adding a parameter etc.. Each of the requirements consists of many elements that could be stated on its own. For simplicity a generalization was made.

4.1.2 Non-functional requirements

Non-functional requirements describe what the system ought to be like. Below there is a list of the main characteristics that define a good implementation of the system.

The product shall:

- be easy to use - to ensure high usability,
- interpret user’s natural English sentences - the system is directed towards English speakers,
- work with non-native speakers - there are many different accents and the system should work with most of them,
- be reliable - system should not crash or lose data,
- not perform many mistakes - to ensure usability and decrease frustration,
- have a way to correct mistakes - mistakes done either by the system or the user should have an easy way of being corrected,
- be intuitive - should require little time to be learned,
- inform about errors and a way to prevent them - user has to be always informed of what is happening,
- quickly respond to the user - to increase productivity,
- give precise feedback - user should be able to understand what has happened, might have an influence on speech recognition,
- must work with Eclipse IDE - system is intended to work as an Eclipse plugin,
- be easily available from within Eclipse Market - acquiring and installation of the system should be easy and almost automatic,
- run on Windows OS,
- run on Linux OS.

4.1.3 Constraints

CodeSpeech has couple of constraints resulting from either its nature or technology used. The first obvious constraint is a need of a microphone. The program is heavily dependent on audio input cannot be used without a way to record voice. Another constraint related to that is a requirement of usage in a quiet environment. In order to increase performance of recognition ambient noise should be decreased to the minimum. One software constraint is Eclipse IDE with JDT (Java development Tools) installed on the workstation as the system is intended to be a plugin. Finally, because of the use of selected speech recognition tool, CodeSpeech requires an Internet connection.

4.2 Use cases

Below one can find enlisted use cases with short description. They depict different actions that can be performed from the user's point of view, such as the creation of the specific code structure, moving the cursor to a given line etc. In Fig. 4.1 a visualization in the form of UML Use Case diagram is presented. The main use cases are:

1. Creating source code - most important part of programming. It involves: e.g.
   - creation of project elements, such as project, package, compilation unit,
   - creation of programming structures, such as variables, methods definitions, invocations, loops, if conditions etc.
2. Modifying source code - due to mistakes or constant development process, modification of source code is inevitable. Examples of modification:
   - name, type changes,
   - values assignment, initialization,
   - addition, removal of method parameters/arguments,
   - deletion of a structure.
3. Local navigation (in file) - placing a cursor in a specific position in a file:
   - vertical navigation (between lines),
   - optional: horizontal navigation (by character/nodes e.g. parameter).
4. Global navigation (between files) - such as:
   - selection of other projects in the workspace,
   - opening another compilation unit for modifications.
5. Calling Eclipse commands - programming is not only writing source code, it also involves other actions, which nowadays are built-in into IDEs. That is why Code-Speech was selected to be an extension of such an environment and provide a way to e.g.:
   - put/enable/disable a breakpoint into a specific line,
   - compile program,
   - run/debug program.

Use Case 1
User/actor name: Programmer
Description: Create code
Use Case Scenario:

1. User speaks
2. SR engine processes the audio signal and returns recognized speech in text
3. Plugin recognizes command
4. Plugin performs operations on AST to create new code
5. Plugin updates IDE

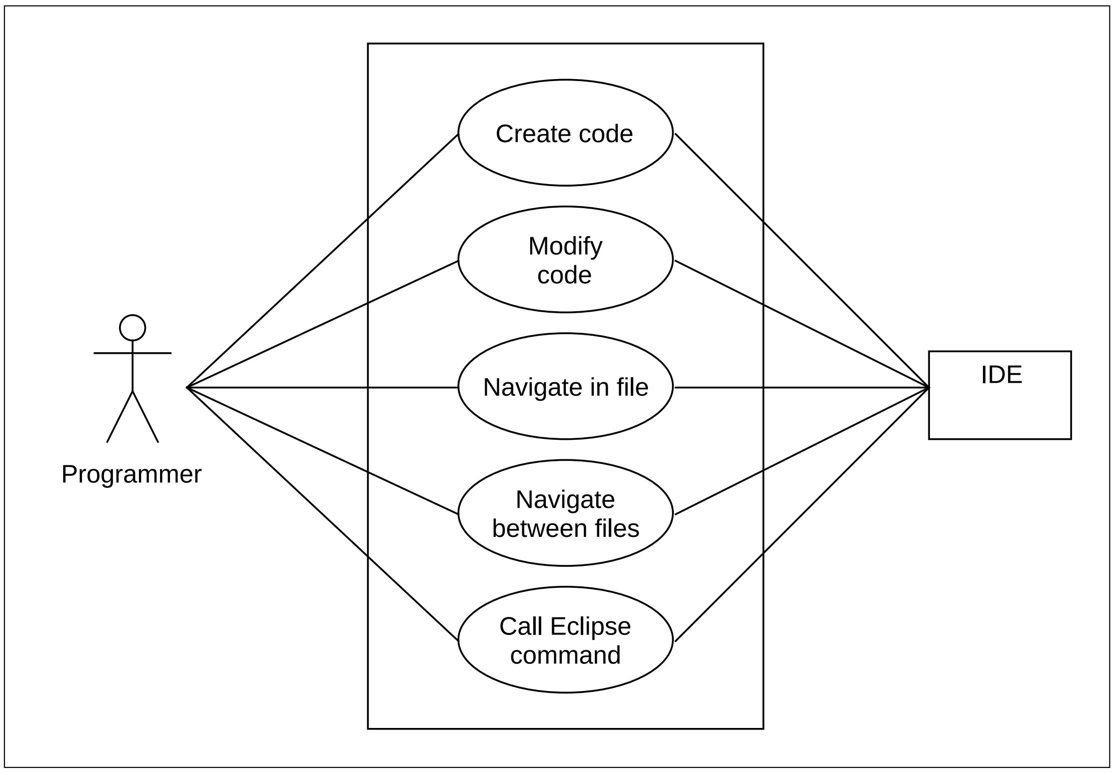


Figure 4.1: Use Case diagram.

Use Case 2
User/actor name: Programmer
Description: Modify code
Use Case Scenario:

1. User speaks
2. SR engine processes the audio signal and returns recognized speech in text
3. Plugin recognizes command
4. Plugin performs operations on AST to manipulate existing code
5. Plugin updates IDE

Use Case 3
User/actor name: Programmer
Description: Navigate in file
Use Case Scenario:

1. User speaks
2. SR engine processes the audio signal and returns recognized speech in text
3. Plugin recognizes command
4. Plugin moves cursor to given position

Use Case 4
User/actor name: Programmer
Description: Navigate between files
Use Case Scenario:

1. User speaks
2. SR engine processes the audio signal and returns recognized speech in text
3. Plugin recognizes command
4. Plugin opens new editor and displays file's content

Use Case 5
User/actor name: Programmer
Description: Call IDE command
Use Case Scenario:

1. User speaks
2. SR engine processes the audio signal and returns recognized speech in text
3. Plugin recognizes command
4. Plugin performs calls IDE's command

As it can be seen, the only thing required by the user to do is to speak the right command. The system then takes care of analyzing the speech and performing the right action.

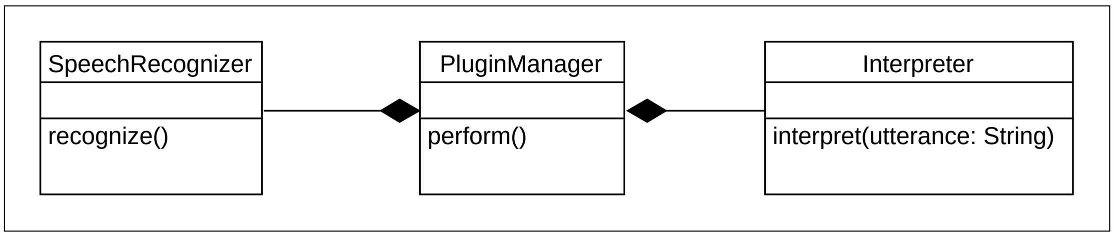
4.3 Design

This section presents some ideas for the architecture of the system and its interface, discusses design choices backing them up with arguments, examples and summarises the advantages and disadvantages of these choices.

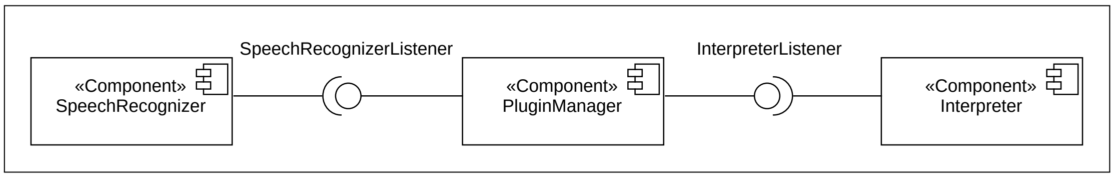
4.3.1 System Architecture

The structure of the system will consist of three main parts. One will be responsible for speech recognition, second for plugin management and the third for utterance interpretation. These are planned to be implemented in three structures: SpeechRecognizer, PluginManager and Interpreter, respectively. Fig. 4.2 presents these components in simple UML diagrams.

The main idea is like this: PluginManager is the main structure of the plugin so it initializes other structures and all necessary tools. This is due to the way Eclipse plugin structure look like. When plugin is on, SpeechRecognizer waits for audio input. When the user stops speaking, audio is treated by the implementation of currently used SR toolkit. When done, PluginManager is notified and receives the recognized utterance in the form of text. PluginManager then delegates the utterance to the Interpreter. Interpreter analyses the text and builds a proper Command object. Once the whole utterance has



(a)



(b)

Figure 4.2: Initial class diagram (a). Component diagram (b).

been walked through, the PluginManager is notified and receives the Command. Then, if everything is set, PluginManager executes the command and performs the action on IDE. The program can be interrupted and turned off at any stage of the recognition. Basic state diagram presenting the workflow is depicted in Fig. 4.3.

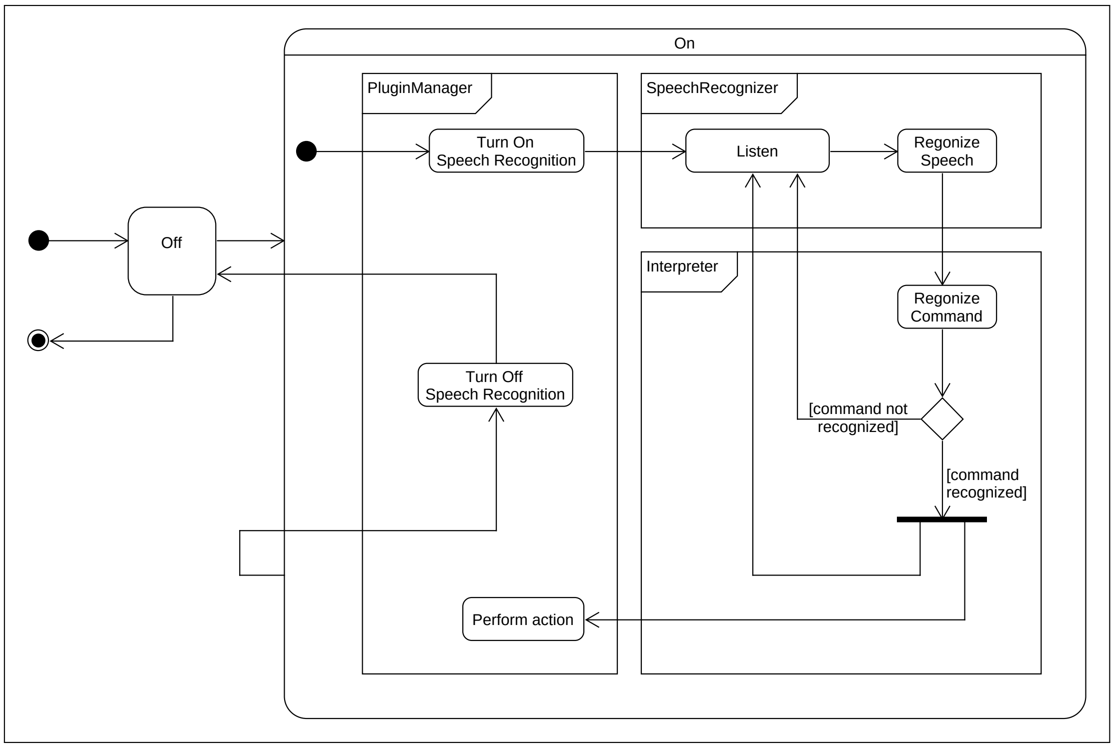


Figure 4.3: State diagram of the program.

The system diagram is presented in Fig 4.4. Microphone does not have to be external, it can be one built into the laptop. Internet connection might be optional, depending on the system recognition tool that is being used. There might be a case in which a cloud service is utilized, and Internet connection is required. The system could be any which can run Eclipse IDE.

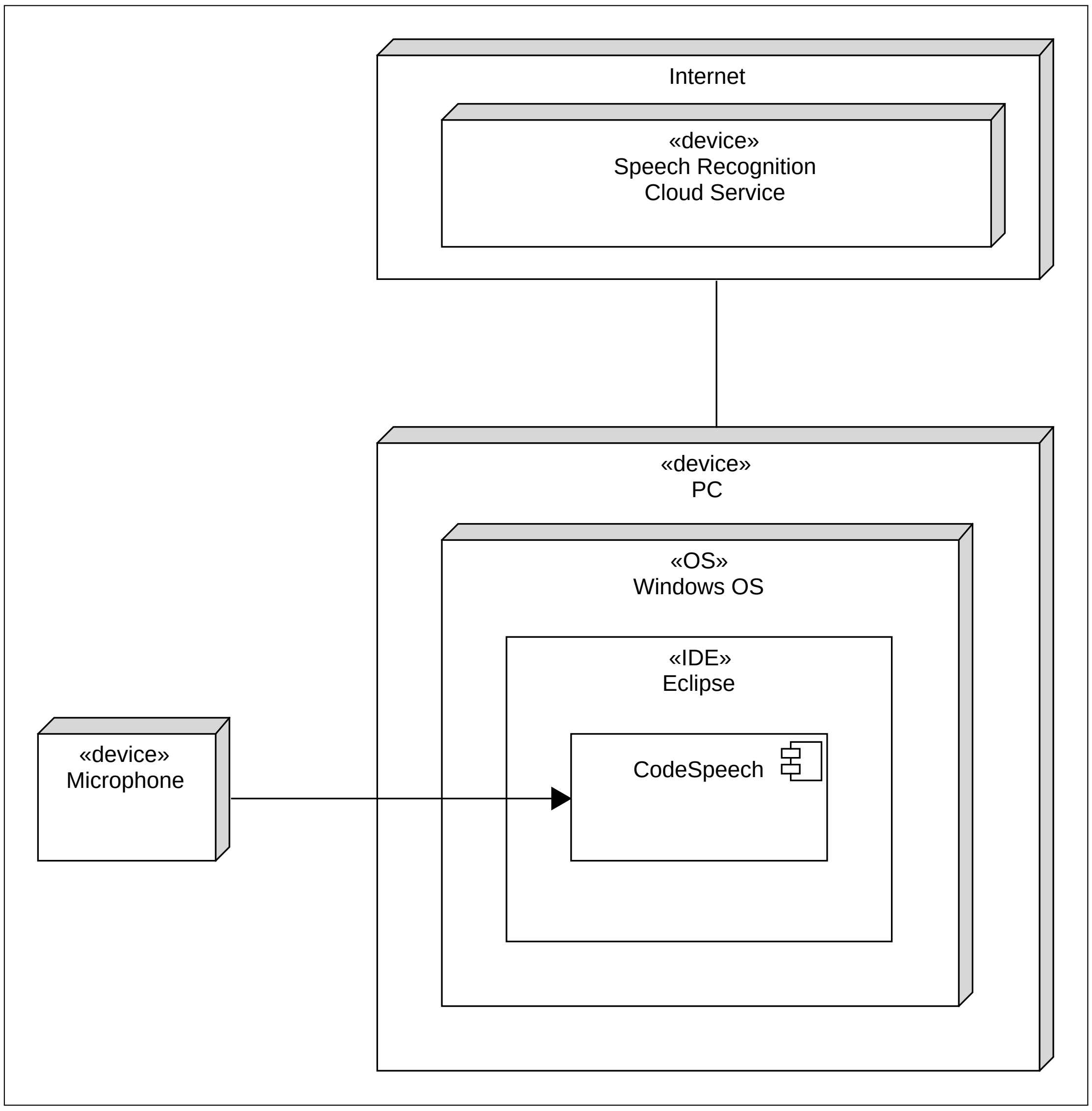


Figure 4.4: System diagram.

4.3.2 Interface

In order to have control over the plugin a button will be put into the toolbar menu. Clicking the button will toggle on and off programming by voice. Location of the button is presented in Fig. 4.5. This method is a temporary solution because it still requires the use of mouse. In the future this way of interaction should be possibly replaced by activating via a specific command to decrease the need of using a mouse to zero. That will require to start up the programming by voice and activate microphone from the initialization of the IDE and wait for a specific voice command, such as e.g. “start voice programming”. Similarly “stop voice programming” would turn the main functionality off.

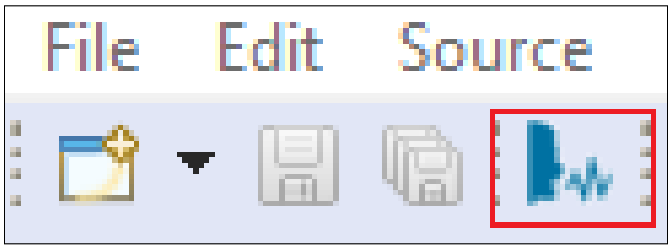


Figure 4.5: Toggle button (in red frame) added to the menu to enable and disable programming by voice.

There exists an alternative way to control the plugin without the need of a mouse, namely a “mouse grid” tool that some of the available SR software, such as Windows SpeechRecognition, provide. Users can activate a grid that divides a screen into nine sections, and by speaking the number of a section where their interest lies, the grid scales and appears in that section as presented in Fig. 4.6. When one cell of a grid lies entirely in the position which they want to perform click on, it can be done simply speak a number of this cell and end with “click” word. That is a general idea behind this function, details might differ from one tool to another. After CodeSpeech is activated, external SR tool can be turned off. Stopping voice programming works analogically.

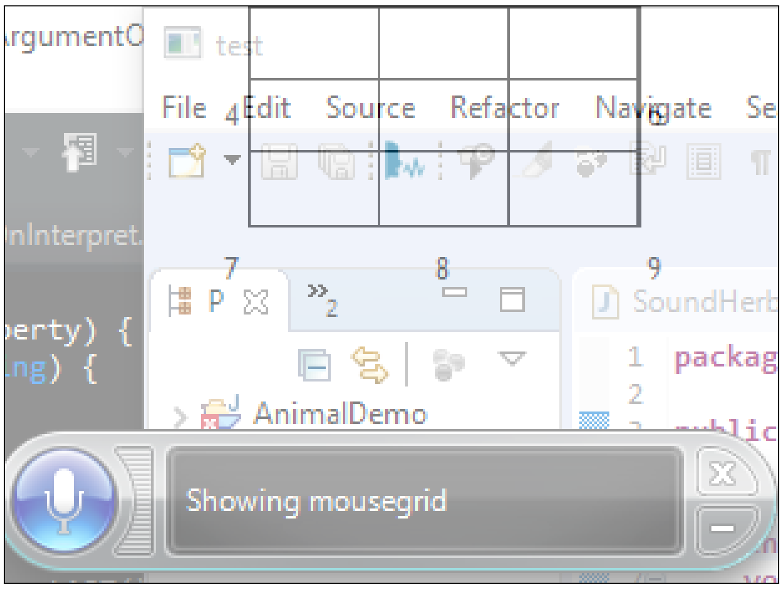
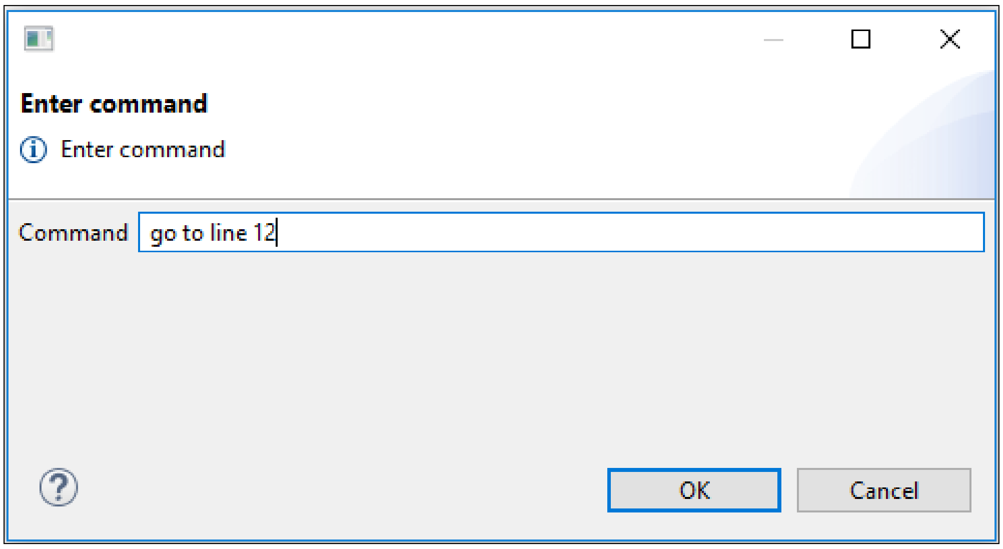


Figure 4.6: Mouse grid tool - alternative to the use of mouse build into Windows 10 SR.

In addition a dialog box was introduced in the first iteration of the project in order to allow for introducing commands via text. This allows skipping the speech recognition phase and is to be used for debugging. This dialog is not planned to be used in the final version of the plugin. Fig. 4.7 depicts the command dialog box.



go to line 12

Figure 4.7: Command Dialog box that allows manual entry of commands.

Chapter 5

Implementation

In this chapter the most important details about the implementation are revealed. The general idea behind the workflow is described and presented in the form of sequence diagrams. Class and package diagrams show basic architecture of the implementation, and the most important structures are discussed with a few interesting source code listings.

5.1 Packages

As could be seen in the previous section, whole plugin is divided into three main parts: plugin management, speech recognition and interpretation. Classes important for each particular part are placed in the respective packages. The package structure can be seen in Fig. 5.1.

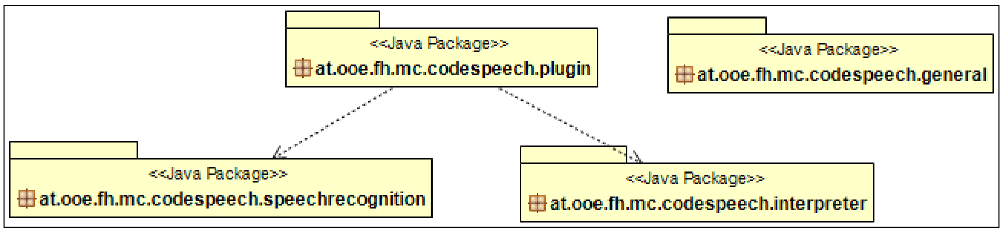

<<Java Package>>
at.ooe.fh.mc.codespeech.plugin
<<Java Package>>
at.ooe.fh.mc.codespeech.general
<<Java Package>>
at.ooe.fh.mc.codespeech.speechrecognition
<<Java Package>>
at.ooe.fh.mc.codespeech.interpreter

Figure 5.1: Basic package structure and dependencies.

5.1.1 *general* package

The *general* package contains elements of the system that are common for every other package. Its subpackage *exceptions* is a place for classes that can be used to throw exceptions. As for now there exist two of those, namely `NotImplementedException` and `InvalidNumberException`. The *events* subpackage consists of `Even<T>` abstract class that is a parent of any other implementation of an event to be handled by `EventHandler`. An example of a child event is `OnInterpretationFinishedEvent` of *interpreter* package. Finally, *utils* subpackage has two static utility classes that allow operations on strings (`StringUtils`) and conversion of word to number (`WordToNumber`).

5.1.2 *plugin* package

The *plugin* package consists of elements that are related to the management of a plugin. It contains the most important `PluginManager` class, as well as static helper classes used for handling of operations performed on text editors, package explorer, abstract syntax tree and for global search functionality. These static classes are named `EditorManager`, `PackageExplorerManager`, `ASTManager` and `Searcher` respectively and are held in *utils* subpackage. Another class contained in *plugin* package is `ToggleSpeechRecognition-Handler`, which turns on or off active listening when menu button is toggled. For debugging purposes an additional dialog box was added to the IDE, which allows to skip speech recognition part and enter the command in the text form. The dialog is implemented as `CommandDialog`. By pressing the OK button an `EnterCommandHandler` is triggered. All three of these are part of the plugin and therefore they can also be found in the subpackage *menu*.

5.1.3 *speechrecognition* package

As the name suggests this package holds implementation of speech recognition elements. The most important one is `SpeechRecognizer` abstract class, which enables usage of many different SR toolkits. Currently there are three different classes utilizing different APIs that inherit from it, namely `Sphinx4SpeechRecognizer`, `PocketsphinxSpeechRe-cognizer` and `GoogleSpeechRecognizer`. These were put into subpackages *cmusphinx* and *googlespeech*. To change between SR technologies a specific `Enum` value of `SREngine-Type` has to be passed into the `SpeechRecognitionFactory`. The factory initializes and returns specified `SpeechRecognizer` object. Selection of SR toolkit is not available from the user side, by default `GoogleSpeechRecognizer` is returned. `SpeechRecognitionLi-stener` interface which provides handling methods for recognition events is also present in this package, together with the event classes. These children of `Event<T>` class are found in *events*. `Microphone` class is used for management of audio recording, therefore is a part of this package.

5.1.4 *interpreter* package

The main classes of this package are `Interpreter`, `InterpreterContext` and `Command`. `InterpreterListener`, which provides a way to handle interpretation events can also be found in here. For now only one such event exists, namely `OnInterpretatioFinished-Event` placed inside *events* subpackage. The *grammar* subpackage is where ANTLR generated classes are being kept. Every keyword listener that extends `BaseKeywordListener` (including `BaseKeywordListener` itself) has been placed in *listeners* subpackage. `Opera-tion` interface together with its implementations can be found in *operations*. They provide methods that consist of logic for performing specific actions, such as creation of a structure, navigation etc. Therefore, according to usage they have been separated into three categories that exist so far: *creation*, *modification* and *navigation*. In order to ensure creation/modification of a proper programming structures, their representation has been introduced in a form of models. They all extend `Model` abstract class and can be found *models* subpackage. Examples of such models are e.g. `ClassModel`, `MethodInvocationModel` or `ConditionalModel`.

5.2 Most important classes and interfaces

In Fig. 5.2 one can find class diagram presenting the most important classes together with their relations. Many of these classes rely on other ones that are not depicted in here to ensure readability of the graph. Some of them will be mentioned, described or presented on another images further on. In the center of the graph with yellow color there is `PluginManager`. On its left side, depicted in green there are classes responsible for interpretation and on the right with blue color, those that take care of speech recognition. For the sake of simplicity only one subclass of `BaseListener`, `Model` and one implementation `Operation` are depicted. It is to show an example of how these classes are related. The relationships of other children are analogous.

5.2.1 PluginManager

`PluginManager` is the heart of the plugin (usually called "Activator" in official documentation of PDE development). It extends `Plugin` class, which is a part of OSGi framework. Its `start` method is called the first time the plugin is initialized, which happens whenever any element of plugin is required for the first time. It is there, where all necessary objects are created. Additionally, `PluginManager` implements two interfaces created by the author, namely `SpeechRecognitionListener` and `InterpreterListener`. As the names suggest, these are the interfaces that allow handling of events related to recognition of speech and text interpretation respectively. During initialization of the plugin `Interpreter`, as well as a `SpeechRecognizer` is created. `PluginManager` also implements `IPartListener2` in order to get notified in case of any change in text editors. So far this is utilized only to keep track of currently opened editor.

5.2.2 SpeechRecognizer

`SpeechRecognizer` is an abstract class and a parent of `Sphinx4SpeechRecognizer`, `PocketsphinxSpeechRecognizer` and `GoogleSpeechRecognizer` which utilize different SR toolkits. It contains `Microphone`, `EventHandler`, `Mode` and `RecognierThread` instances, and also a list of `SpeechRecognitionListeners`. It provides methods to start and stop audio listening. As it is implemented now, three modes of recognition are possible: continuous speech, based on grammar and based on keyphrases. Only CMUSphinx recognizers provide a way for grammar-based recognition, and only PocketSphinx API can recognize in keyphrase mode. Implementation uses only continuous speech as for now. If any new SR toolkit is to be used, simply a new class has to inherit `SpeechRecognizer` and use its API in there. `Microphone` is responsible for gathering audio data, `Mode` is an enumeration type and its instance helps to keep track of current recognition mode.

5.2.3 RecognizerThread

`SpeechRecognizer's` inner class `RecognizerThread` extends `Thread` and it is in its `run` method where the main process of recognition is performed. This class has to be extended as well when new SR toolkit is being implemented. The most important to override are three methods: `beforeRecognition`, `recognize` and `afterRecognition`. Start-

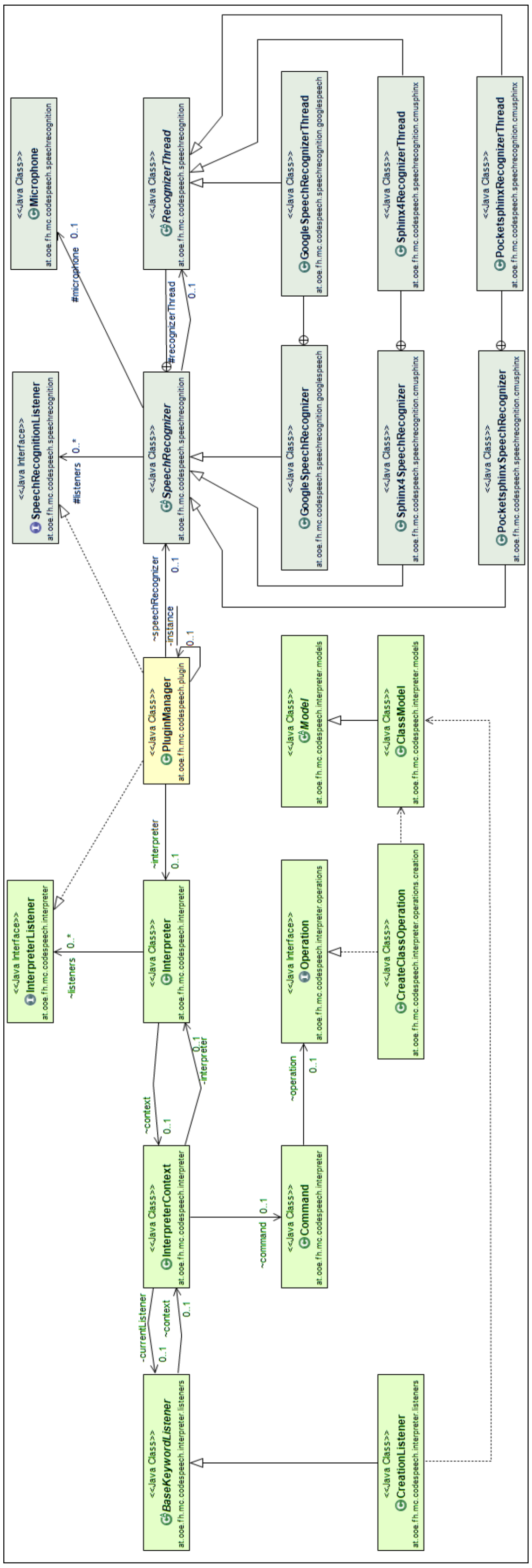


Figure 5.2: Class diagram representing the most important classes and their relations.

ing from the beginning, `beforeRecognition` is where all operations that are mandatory just before the recognition are performed. In `recognize` method the actual recognition logic is done. Finally, `afterRecognition` is where all of the post recognition operations, such as changing state to finished or breaking a connection need to be placed to avoid problems in future recognition.

5.2.4 Interpreter

Next on the list is the `Interpreter` class. It contains `InterpreterContext`, `EventHandler` and a list of `InterpreterListener` instances. Its main functionality is in `interpreter` method, which is being called upon when the recognition is done. It receives a text in a form of `String`. If the text is not empty, everything needed for parsing is set up, together with generated by ANTLR tool `GrammarParser` and `GrammarLexer`. Once everything is ready, `ParseTreeWalker` walks through a tokenized tree. Details are shown in Program 5.1. Once `finish` method is called by the context, `OnInterpretationFinishedEvent` is posted.

Program 5.1: Start of interpretation procedure.

```
public void interpret(String utterance) {
  if(!utterance.isEmpty()) {
    GrammarLexer lexer = new GrammarLexer(CharStreams.fromString(utterance));

    CommonTokenStream tokens = new CommonTokenStream(lexer);
    GrammarParser parser = new GrammarParser(tokens);

    ParseTreeWalker walker = new ParseTreeWalker();

    walker.walk(context.getCurrentListener(), parser.command());
  }
}
```

5.2.5 InterpreterContext

This class was created to gather all important data throughout the process of interpretation. Its instance is passed between listeners when they change. It contains many fields such as `boolean isAbstract` or `String simpleType` that are being set when a keyword, such as “abstract” or “simpleType” respectively, was detected and the corresponding method of a listener called. Once it is known what operation is to be performed and which of these properties are important, proper object is created using these properties. Additionally `InterpreterContext` contains of `Command` object, which at the proper moment is set and at the end of interpretation process passed as an argument to the listeners by the event handler.

5.2.6 BaseKeywordListener

Inheriting from ANTLR generated `GrammarBaseListener`, `BaseKeywordListener` is an abstract parent of every other listener to be used for grammar analysis of the command.

It provides common functionality for other listeners like building up `Command`, canceling if something went wrong (going back to initial state) or finalizing the process. It provides an opportunity of overriding any method that is triggered when a specific token is entered or exited. An example of such method is depicted in Program 5.2. It shows a method of `SelectionListener` activated after the keyword “select” was detected. Once keyword “package” is found, `enterPackageKeyword` method of this listener is triggered and a proper model together with operation is set up.

Program 5.2: Entering “packageKeyword” token in `SelectionListener`.

```
@Override
  public void enterPackageKeyword(PackageKeywordContext ctx) {
    super.enterPackageKeyword(ctx);

    changeProperty(new PackageModel());
    changeOperation(new SelectPackageOperation());
  }
```

5.2.7 Command

`Command` is a simple but important class. It has two objects that are being set during the interpretation. First one is an instance of `Operation` and the second is `Object property`. When the command is executed, `perform` method of operation is called with the `property` object passed as argument.

5.2.8 Operations

`Operation` is an interface and each class that agrees to fulfill its contract has to implement `perform` method. Each operation is different, therefore each is implemented separately. Examples of operations are e.g. `CreateClassOperation` or `ChangeReturnType-Operation` or `SelectAndOpenClassFileOperation`. An example of yet another operation is given in Program 5.3.

5.2.9 Models

As mentioned before, in order to create a proper structure with correct properties models were introduced. Each model is different, depending on which AST node or program structure it represents. All models inherit a getter and setter of a `String phrase`. The reason for that is to avoid huge switch statement and to set names of certain structures, set conditions into others etc. in a unified way.

5.3 Workflow

Below, the most important parts of the working cycle are presented and described in detail. The first important part of a workflow is speech recognition, the second is interpretation. For better understanding of the activities, each of them is depicted in

Program 5.3: `AddArgumentOperation` class. It can be seen that operation classes contain of only one method and nothing else. Operations use utility methods provided by `ASTManager` to perform changes to AST and by `EditorManager` to update compilation unit and place cursor into the new position.

```java
public class AddArgumentOperation implements Operation {

  @Override
  public void perform(Object property) throws Exception {
    if(property instanceof String) {
      ASTNode node = ASTManager.getNextNodeOfType(
          ASTManager.currentNode, MethodInvocation.class);
      if (node != null && node instanceof MethodInvocation) {
        AST ast = node.getAST();
        ASTRewrite rewriter = ASTRewrite.create(ast);
        ListRewrite listRewrite = rewriter.getListRewrite(
            node, MethodInvocation.ARGUMENTS_PROPERTY);

        MethodInvocation methodInvocation = (MethodInvocation) node;
        SimpleName name = ast.newSimpleName((String) property);
        listRewrite.insertLast(name,  null);

        EditorManager.updateCompilationUnit(rewriter.rewriteAST());
        EditorManager.moveToNode(name);
      }

    }
  }
}
```

the form of sequence diagrams. Implementation of the most important part of speech recognition is also enlisted.

5.3.1 Speech recognition

In Fig. 5.3 a sequence diagram presenting speech recognition process is shown. Once plugin is turned on (via toggle of menu button) speech recognition class starts its recognition thread. At the beginning microphone recording is started and `beforeRecognition` method called. Afterwards, functionality loops until it is interrupted. The first thing inside the loop is a check for timeout. If it did occur, an `OnTimeoutEvent` is send, if not, an input stream of a `Microphone` object is read. If audio was successfully read, `recognize` method is called. At the end of successful recognition `OnResultEvent` should be posted. The main loop is surrounded by try-catch-finally block to catch any occurring exceptions that might be thrown in the process and to ensure that `afterRecognition` is called even if something went wrong. At the end microphone has to stop its recording. The full implementation of the process can be found in Program 5.4.

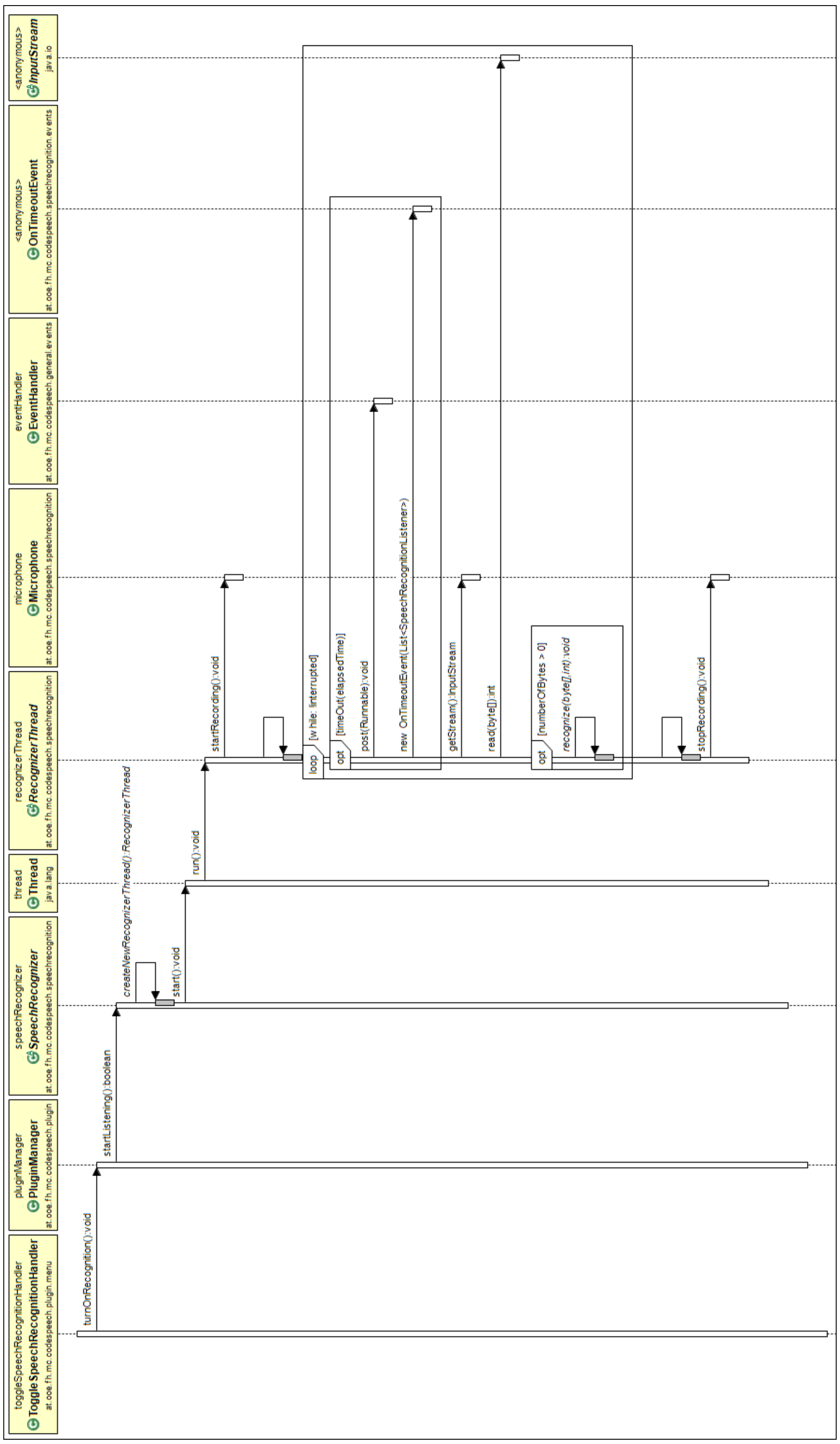

toggleSpeechRecognitionHandler
ToggleSpeechRecognitionHandler
at.ooe.fh.mc.codespeech.plugin.menu
pluginManager
PluginManager
at.ooe.fh.mc.codespeech.plugin
speechRecognizer
SpeechRecognizer
at.ooe.fh.mc.codespeech.speechrecognition
thread
Thread
java.lang
recognizerThread
RecognizerThread
at.ooe.fh.mc.codespeech.speechrecognition
microphone
Microphone
at.ooe.fh.mc.codespeech.speechrecognition
eventHandler
EventHandler
at.ooe.fh.mc.codespeech.general.events
<anonymous>
OnTimeoutEvent
at.ooe.fh.mc.codespeech.speechrecognition.events
<anonymous>
InputStream
java.io
turnOnRecognition():void
startListening():boolean
createNewRecognizerThread():RecognizerThread
start():void
run():void
startRecording():void
loop
[while: !interrupted]
opt
[timeOut(elapsedTime)]
post(Runnable):void
new OnTimeoutEvent(List<SpeechRecognitionListener>)
getStream():InputStream
read(byte[]):int
opt
[numberOfBytes > 0]
recognize(byte[],int):void
stopRecording():void

Figure 5.3: Sequence diagram showing process from start to speech recognition.

Program 5.4: RecognizerThread's run method.

```
@Override
  public void run() {
    long startTime;
    long elapsedTime;

    microphone.startRecording();

    beforeRecognition();

    startTime = System.currentTimeMillis();

    try {
      while(!interrupted) {
        elapsedTime = System.currentTimeMillis() - startTime;

        if(timeOut(elapsedTime)) {
          eventHandler.post(new OnTimeoutEvent(listeners));
          break;
        }

        byte[] bytes = new byte[BUFFER_SIZE];
        int numberOfBytes = microphone.getStream().read(bytes);

        if(numberOfBytes > 0) {
          recognize(bytes, numberOfBytes);
          startTime = System.currentTimeMillis();
        }
      }
    } catch (Exception exception) {
      exception.printStackTrace();
    } finally {
      afterRecognition();
      microphone.stopRecording();
    }

  }
```

5.3.2 Interpretation

Once the sentence is recognized, an `OnResultEvent` event is triggered and the utterance is sent to all listeners. The main one is `PluginManager` class, which delegates recognized text to `Interpreter`. An example of interpretation procedure is depicted as sequence diagram in Fig. 5.4. Once `interpret` method of `Interpreter` object is called, it initializes all the necessary objects needed for ANTLR parsing and starts walking through the parsed tokens. Interpretation always starts with an `InitialListener` (all listeners extend `BaseKeywordListener`), which first listens to the keyword determining operation type and then delegating parsing to another listener accordingly. Listeners can be viewed as the state, which provides different functionality depending on the current situation. Once the listener is changed the interpretation continues. During the whole process, `InterpreterContext` object is being built up, together with its `Command` object. Once the whole text has been analysed, an `OnInterprataionFinished` event is triggered and `InterpreterListeners` are notified. Here again, `PluginManager` is a main subscriber. Once receiving a notification, `PluginManager` performs specific action using generated `Command`.

5.3.3 Performing operation

Operations have different responsibilities depending on action they need to execute. Some modify AST by adding new node, deleting or changing one, others select an element in the package explorer or navigate in the currently open editor. In order to perform its task on IDE `Operation` has to use Eclipse API. Some common functionality has been derived and placed into utilization classes such as `EditorManager`, `ASTManager` and `PackageExplorerManager`. In Fig. 5.5 a sequence diagram showing the workflow of inserting new *if statement* is shown as an example.

5.4 Implemented features

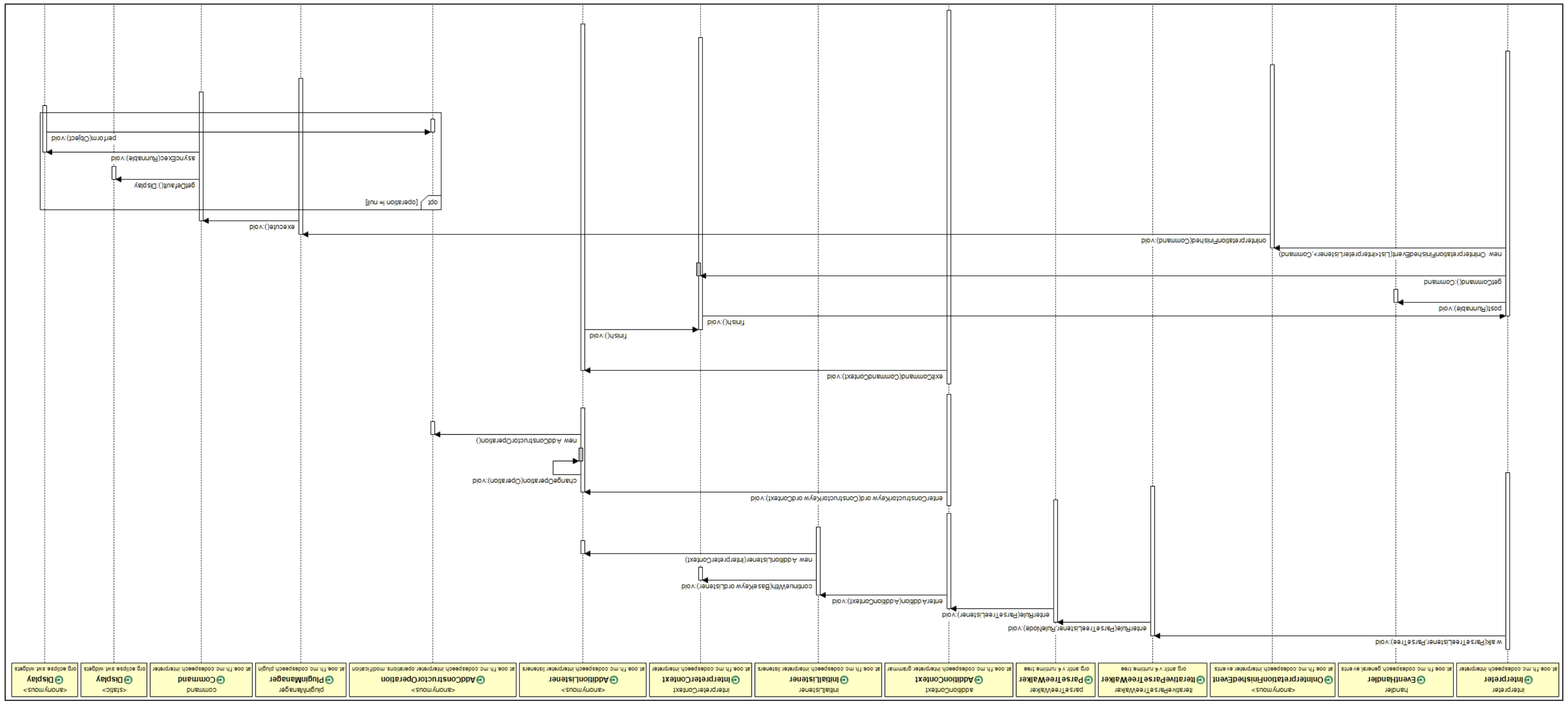
At the current state CodeSpeech provides a limited amount of functionalities. In this section their a complete list can be found. Features have been divided into three groups: creation, modification and navigation and they are presented in Tab. 5.1, Tab. 5.2 and Tab. 5.3 respectively. Some of the most interesting features are described in more detail in the following subsections.

5.4.1 Error Correction

Currently the only error correction measure that works is inside the text editor. Since programming is mostly happening in it, it therefore covers most of the cases. By giving the command “undo last” the last performed action will be reversed. It can come in handy at times, because the alternative is to delete the whole structure and create it anew. This of course would add to delay and frustration of the user. Unfortunately, when a wrong element is created in the package explorer, such as a project, package or a compilation unit with wrong name it has to be removed and the process repeated.

5.4.2 Free Speech Mode

Unfortunately, because of time limits the current state of the project does not allow for creation of the whole program from scratch. In some cases the use of mouse and keyboard is inevitable. In order to limit this need to the absolute minimum, a special free speech mode was added to the implementation. This mode is the latest addition and is entirely experimental. It allows to enter text as it is recognized, with a small modifications to the text, such as changing it to lower case and removing the spaces between words separated by periods (which is supposed to make up for the lack of nested OOP calls, such as `System.out.println()`). This mode could be turned on via “enter free speech” command, and analogically, exited by saying “exit free speech”. To ensure easy error correction in case of wrong recognition, a possibility to reverse last change was kept for this mode.



Interpreter
EventHandler
OnInterpretationFinishedEvent
IterativeParseTreeWalker
ParseTreeWalker
AdditionContext
InitialListener
InterpreterContext
AdditionListener
AddConstructorOperation
PluginManager
Command
Display
Display
walk(ParseTreeListener,ParseTree):void
enterRule(ParseTreeListener,RuleNode):void
enterRule(ParseTreeListener):void
enterAddition(AdditionContext):void
continueWith(BaseKeywordListener):void
new AdditionListener(InterpreterContext)
enterConstructorKeywordContext(ConstructorKeywordContext):void
changeOperation(Operation):void
new AddConstructorOperation()
exitCommand(CommandContext):void
finish():void
finish():void
post(Runnable):void
getCommand():Command
new OnInterpretationFinishedEvent(List<InterpreterListener>,Command)
onInterpretationFinished(Command):void
execute():void
opt
[operation != null]
getDefault():Display
asyncExec(Runnable):void
perform(Object):void

Figure 5.4: Sequence diagram showing interpretation from start of walking through parsed text to execution of operation.

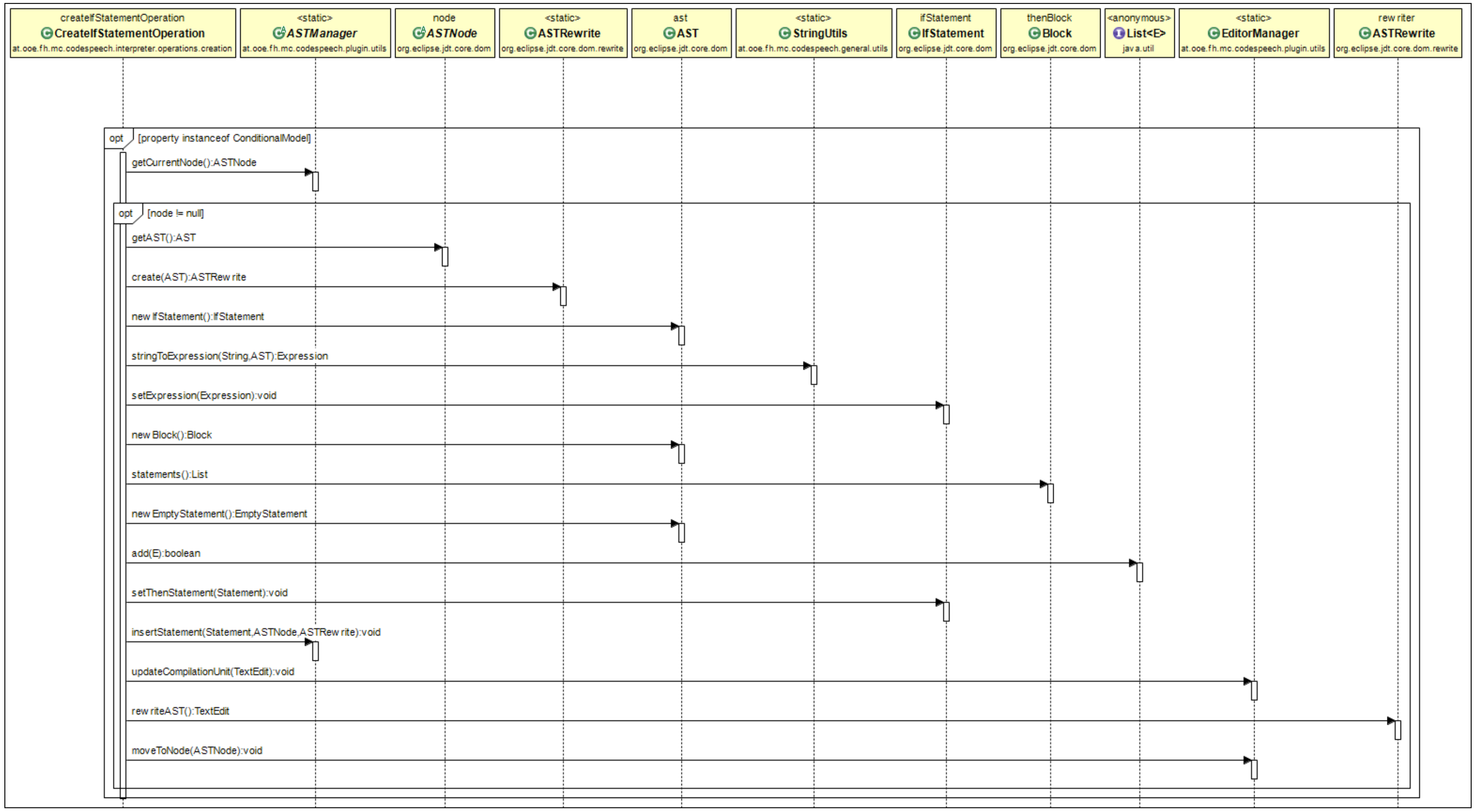
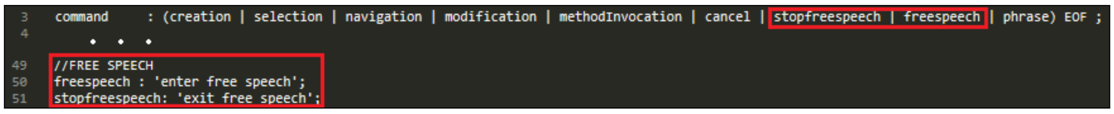

createIfStatementOperation
CreateIfStatementOperation
at.ooe.fh.mc.codespeech.interpreter.operations.creation
<static>
ASTManager
at.ooe.fh.mc.codespeech.plugin.utils
node
ASTNode
org.eclipse.jdt.core.dom
<static>
ASTRewrite
org.eclipse.jdt.core.dom.rewrite
ast
AST
org.eclipse.jdt.core.dom
<static>
StringUtils
at.ooe.fh.mc.codespeech.general.utils
ifStatement
IfStatement
org.eclipse.jdt.core.dom
thenBlock
Block
org.eclipse.jdt.core.dom
<anonymous>
List<E>
java.util
<static>
EditorManager
at.ooe.fh.mc.codespeech.plugin.utils
rewriter
ASTRewrite
org.eclipse.jdt.core.dom.rewrite
opt
[property instanceof ConditionalModel]
getCurrentNode():ASTNode
opt
[node != null]
getAST():AST
create(AST):ASTRewrite
new IfStatement():IfStatement
stringToExpression(String,AST):Expression
setExpression(Expression):void
new Block():Block
statements():List
new EmptyStatement():EmptyStatement
add(E):boolean
setThenStatement(Statement):void
insertStatement(Statement,ASTNode,ASTRewrite):void
updateCompilationUnit(TextEdit):void
rewriteAST():TextEdit
moveToNode(ASTNode):void

Figure 5.5: An example of operation workflow for operation modifying AST. The use of utilization static classes such as `StringUtils`, `ASTManager` and `EditorManager`.

5.5 Addition of new functionality

Once the design of the basic structure was established, adding new functionalities became quite easy. An example will be given on one of the latest features added into the program, which allows entering text straight into the editor.

Basically whenever a new feature is to be implemented, what has to be done first is the addition of a new parser rule, with which the action is to be called, to the grammar. Fig. 5.6 depicts newly inserted phrases. When modification is done and the grammar is ready, ANTLR files have to be generated as described in the official documentation (or in *Tools* chapter).



```
command     : (creation | selection | navigation | modification | methodInvocation | cancel | stopfreespeech | freespeech | phrase) EOF ;

     • • •

//FREE SPEECH
freespeech : 'enter free speech';
stopfreespeech: 'exit free speech';
```

Figure 5.6: Part of grammar. The text in red frame has been added to provide new commands.

For every new parser rule added, the generated files provide new methods for entering and exiting the rule during parsing. In this example such methods are `enterFreespeech`, `exitFreespeech`, `enterStopfreespeech` and `exitStopfreespeech`. These are inserted into `GrammarListener` interface and will become available to every other custom listener that implements it (in the current implementation all listeners do through the inheritance of `BaseKeywordListener`). Whenever it is necessary, a new listener ought to be created. This step can be skipped if one of the existing listeners can be used, e.g. if creation of new structure is to be added, an already existing `CreationListener` will most likely be extended. That is not the case now and so `FreeSpeechListener` must be created. It is supposed to be activated on `enterFreespeech`, so this method is overridden in the `InitialListener`. In there the current listener (state) of the `InterpreterContext` is changed to the new one like shown in Program 5.5.

Program 5.5: Replacing the current listener to `FreeSpeechListener` triggered by "enter free speech" command.

```
@Override
public void enterFreespeech(FreespeechContext ctx) {
  super.enterFreespeech(ctx);
  context.continueWith(new FreeSpeechListener(context));
}
```

`FreeSpeechListener` needs to do three things: put text into the editor, be able to reverse changes and go back to the initial state. Adding new text to the editor is new functionality, therefore a new `Operation` is required. This operation should take raw text as an input, rework it and add it to the editor. The resulting `FreeSpeechOperation` can be seen in Program 5.6.

The next feature available from this state is change reversal. The `UndoOperation` already exists and is simply be reused. Finally, on command "exit free speech" the state is

Program 5.6: Implementation of new functionality, which works on string and puts it into the current line in the editor.

```
public class FreeSpeechOperation implements Operation {
  @Override
  public void perform(Object property) throws Exception {
    if(property instanceof String) {
        EditorManager.enterText(((String) property).replace(". ", "."));
    }
  }
}
```

changed to the initial one (meaning changing the current listener to `InitialListener`). Property and operation are cleared. Whole `FreeSpeechListener` can be seen in Program 5.7. That concludes the process of adding new functionality. Details might vary depending on how complex is the command, how many parser rules it consists and so on. Sometimes using more than one listener might be a viable solution to perform a task.

Program 5.7: Implementation of free speech state that is being triggered by three different commands.

```
public class FreeSpeechListener extends BaseKeywordListener {
    ...
  @Override
  public void enterPhrase(PhraseContext ctx) {
    super.enterPhrase(ctx);
    changeProperty(ctx.getText());
    changeOperation(new FreeSpeechOperation());
  }
  @Override
  public void enterStopfreespeech(StopfreespeechContext ctx) {
    super.enterStopfreespeech(ctx);
    changeProperty(null);
    changeOperation(null);
    context.continueWith(new InitialListener(context));
  }
  @Override
  public void enterUndo(UndoContext ctx) {
    super.enterUndo(ctx);
    changeOperation(new UndoOperation());
  }
}
```

Table 5.1: List of functionalities from creation group.

Functionality	*Details*	*Example command*
Create Java project	Given name is in format of Pascal case.	create project project name
Create package	Given name is in Java convention format.	create package package name
Create class	Given name is in Java format of Pascal case.	create class class name
Create class / interface file	Given name is in Java format of Pascal case. Can be abstract, final, with access modifier.	create abstract class class name
Create method body	Given name is in Java format of Camel case. Can be abstract, static, final, with access modifier.	create public static method main method
Create variable	Given name is in Java format of Camel case. Can be static, final, with access modifier. So far allows primitive and simple types and arrays thereof.	create integer count create array list named list create array of string named string list
Create if statement	Allows for simple prefix, infix and postfix conditions. Does not allow method invocations, complex conditions or String comparison.	create integer count create if count is greater than zero
Create else statement	Allows adding else of else if statement. Conditions restricted as for if statements.	create else
Create while loop	Conditions restricted as for if statements.	create while not true
Create method invocation	Allows to call a method on an object, class or without.	call get state on out call print on class integer call to string

Table 5.2: List of functionalities from modification group.

Functionality	*Details*	*Example command*
Add argument to method call	Possible is adding one only argument at a time.	add argument name
Add constructor	Adds constructor to the class. Allows only public constructor (by default).	add constructor
Add parameter to method body	Possible is adding only one argument at a time. Allows primitive type, simple type and array thereof.	add parameter double result add parameter array of class name named parameter name
Assign value	So far possible is only assignment of a simple name to simple name.	simple name assign another simple name
Change return type	Changes return type of the method body. Allows primitive type, simple type and array thereof.	change return type to double
Delete argument	Deletes argument on a given position. Only one at a time is possible.	delete argument two
Delete node	Deletes current AST node. It can be variable definition, else statement or whole method.	delete line
Delete parameter	Deletes parameter on a given position. Only one at a time is possible.	delete parameter one
Delete project	Requires a project to be selected.	delete project
Delete package	Requires a package to be selected.	delete package
Delete class/interface file	Requires a class/interface to be selected.	delete class
Extend class	Adds extends statement to the class.	extend parent class
Implement interface	Adds implements statement to the class	implement interface
Free speech	Allows addition of recognized text as it is.	enter free speech text to put exit free speech
Initialize variable	Adds initialization depending on type. Can be a number, simple type or text in quotes for String.	initialize to text
Add return statement	Adds return statement for return type. Allows numbers and simple types.	return variable name
Undo last action	Reverts last change in the editor.	undo last

Table 5.3: List of functionalities from navigation group.

Functionality	*Details*	*Example command*
Line navigation	Moves cursor to given line number.	go to line 10 go to 5
Select project	Selects project in package explorer.	select project project name
Select package	Selects package in package explorer. Project has to be selected first. In case of subpackage only its name can be given (e.g. select package models can select animals.models).	select package package name
Select and open class or interface	Selects in package explorer and opens in new editor class/interface file.	select interface interface name select class class name

Chapter 6

Evaluation

After close to 400 hours spent on design and implementation, the finished prototype was created. It still does not allow to create a whole program entirely by voice, but it is ready for its first evaluation. This evaluation is not a user study per se, because it was done only by the author himself. It is therefore only an initial test of a prototype's functionality which is supposed to give an idea if the project gets closer to reaching the goal and if there is a potential worth in continuation of work. The observations arising from the test's result can also point a direction in which further development should go to, as well as it can be a foundation for a real user study that is planned in the near future.

6.1 Approach

The rationale behind the evaluation is to find out if the created system can be introduced as an alternative for the standard way of fulfilling the task of writing programs and therefore decrease the amount of mouse and keyboard usage. In order to do so, a test was carried out, the goal of which was recreation of a simple Java program using two input methods and compare the results. The two methods under test were: a common one based on typed input and cursor control, and a second one relying on voice. Because CodeSpeech is still just a prototype and not a finished product, its evaluation required partial usage of peripheral input devices in places where the necessary functionality was lacking.

The Java program that was to be created during the test was very simple, consisting of one interface, one abstract class implementing this interface, three classes extending the abstract one, and a main class which initialized three instances, called their methods and printed resulting text as an output to the console. All program elements can be found in Appendices B.1 - B.6, but to briefly get an idea of its complexity the class diagram is presented in Fig 6.1.

The workstation on which the experiment has been conducted was Lenovo Y510P laptop with Intel i7 2.40GHz processor and built in keyboard and an external optical mouse Dell DZL-MS111-L(B). The station was running Windows 10 operating system and Eclipse IDE of version 2019-06 (4.12.0). Voice recording was done by an internal microphone built into the laptop. The test was performed in English language by a

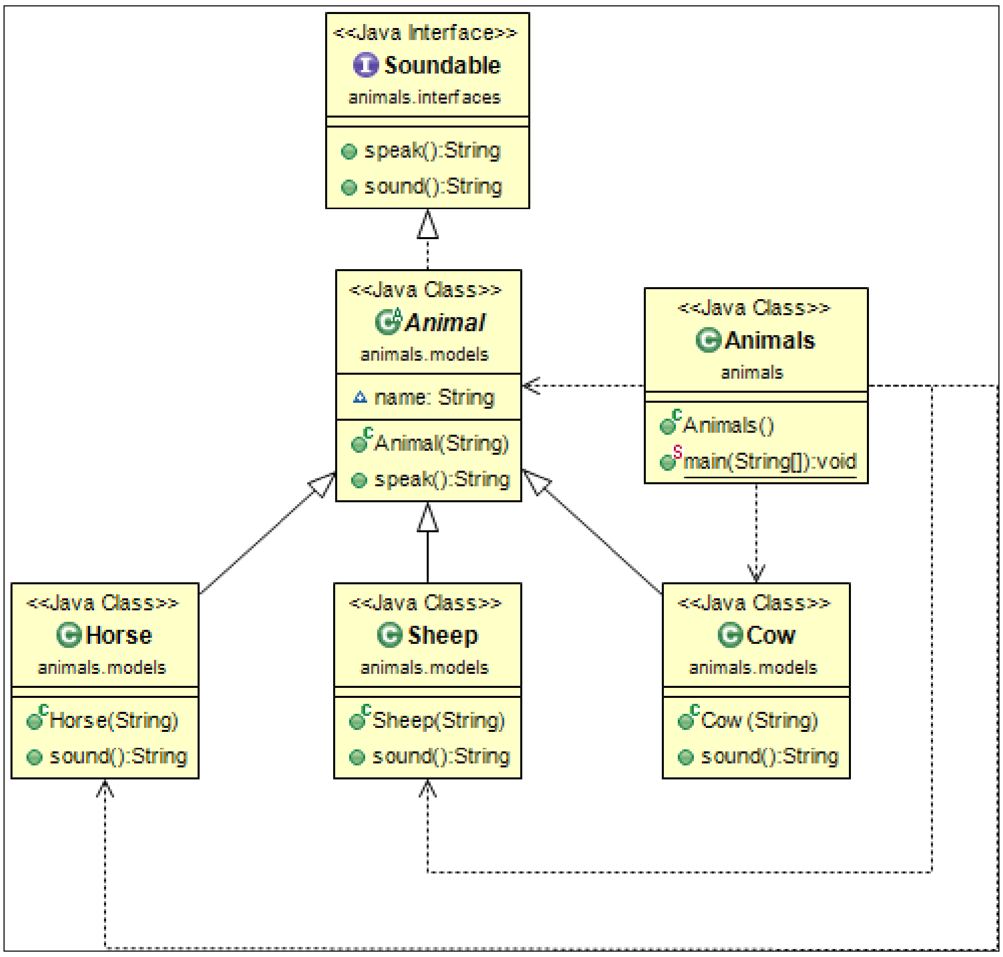


Figure 6.1: Class diagram of test program used to compare both input methods.

non-native speaker.
The software tool used to record an activity of mouse and keyboard was Mousotron 12.1 [1]. Mousotron allows to monitor different types of activities. The ones that were selected as relevant for this evaluation were:

- running time,
- cursor distance,
- number of keystrokes,
- number of left button clicks,
- number of right button clicks,
- number of middle button clicks,
- number of double clicks,
- mouse wheel scrolls,
- and idle time.

[1]http://www.blacksunsoftware.com/mousotron.html

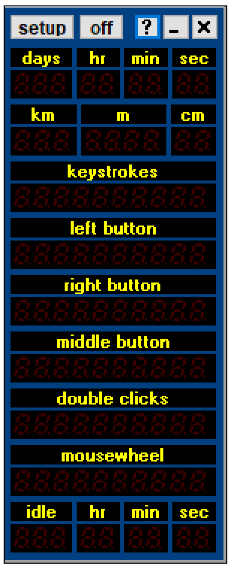


Figure 6.2: Mousotron program used to measure mouse and keyboard activity.

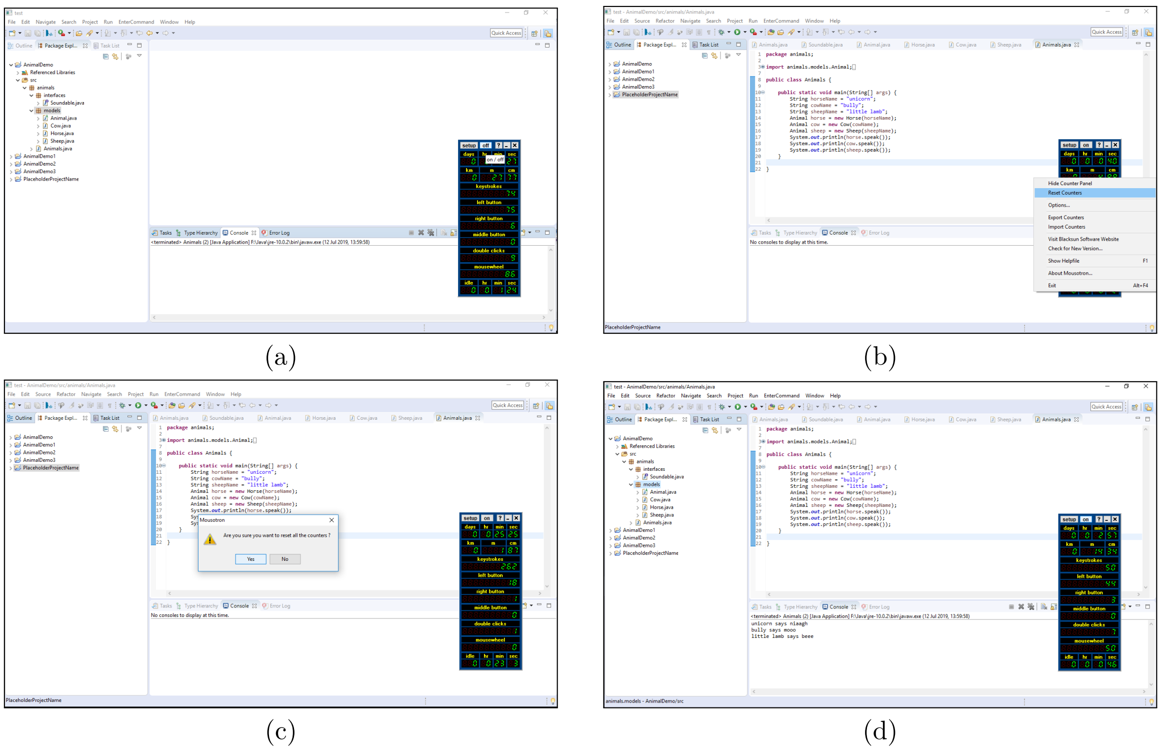
Running time is measured from the beginning to the end of recording. The timer starts or stops by pressing on/off button according to the current state. Idle time measures the time in which no activity of any of the peripheral devices was detected. The delay of starting the counter was set to zero. By subtracting idle time from running time we can get the real time of keyboard and mouse usage (without distinction between them). Cursor distance monitors what distance was traveled by the cursor in either metric or English system. The distance is calculated using the size of the display and the current resolution of the screen. The display used during the test was an 18,5 inches Samsung S19A300N with resolution of 1366x768. Although this value does not give the physical distance traveled by the mouse, it still adds an important information if used in comparison between two sessions. The relation between this distance and the physical distance is proportional if the same mouse sensitivity is used. The rest of the metrics are self explanatory. Single clicks of left, right and middle button were combined to provide a single measure. The layout of the Mousotron application is presented in Fig. 6.2.

In addition to input devices' metrics, another one was introduced in order to evaluate the performance of the system, namely Command Error Rate (CER). CER was measured as a ratio of the number of incorrectly interpreted (and recognized) commands (IC) versus the number of all spoken commands (N). Its formula therefore looks like this:

$$CER = \frac{IC}{N}.$$

CER can be viewed as a derivative of WER of which usage did not make sense in this context. Presenting calculated WER would simply be an evaluation of speech recognition tool, and that was already done in the *Tools* chapter. Instead what was preferred, was an evaluation of the performance of the whole cycle of the system, which includes interpretation, as well as recognition. The latter was made in such a way to ignore some of the words that could be in some cases accidentally introduced by SR, such as articles. Those cases should not be counted as an error if the final command was performed correctly, thus the need for CER.

Once the test program was prepared and the recording software set up, the evaluation could be started. At first the program was done using the standard method, followed by the voice control method. In order for both methods to be comparable, the management of the recording had to be unified to minimize noise. The detailed description of the process backed up by screenshots can be seen in Fig. 6.3.



```
package animals;

import animals.models.Animal;

public class Animals {

    public static void main(String[] args) {
        String horseName = "unicorn";
        String cowName = "bully";
        String sheepName = "little lamb";
        Animal horse = new Horse(horseName);
        Animal cow = new Cow(cowName);
        Animal sheep = new Sheep(sheepName);
        System.out.println(horse.speak());
        System.out.println(cow.speak());
        System.out.println(sheep.speak());
    }

}
```

Are you sure you want to reset all the counters ?

```
<terminated> Animals (2) [Java Application] F:\Java\jre-10.0.2\bin\javaw.exe (12 Jul 2019, 13:59:58)
unicorn says niaagh
bully says mooo
little lamb says beee
```

Figure 6.3: At first Mousotron and Eclipse IDE had to be both opened, visible and set to proper positions. Mousotron was set to be always on the top layer and placed in the right bottom corner of the screen to prevent covering anything considered important, to avoid the necessity of moving it during the test. This way data noise could be reduced. Once ready, the mouse and keyboard recording was started using an on/off button (a). In order to further reduce noise the counters were reset to zero via "Reset Counters" option (b). This made a dialog box to appear, and the counters where restarted once confirmed (c). This ensured that the initial position of the cursor was the same for both methods. After confirmation programming could begun. Finish was marked by the successful run of the program with the correct output printed into console. To stop recording, the on/off button had to be pressed again (d).

6.2 Results

The results of the evaluation of two different input methods are presented in Tab. 6.1. In the table one can see the scores of previously selected measures. The first column

Table 6.1: Results of evaluation.

	Method 1	*Method 2*
Total time	5 min 57 s	24 min 54 s
Device usage time	5 min 06 s	1 min 55 s
Cursor distance	887 cm	154 cm
No. of keystrokes	876	262
No. of clicks	84	17
No. of double clicks	19	1
Mouse wheel scrolls	0	0

titled as "Method 1" represents the standard way of providing input through mouse and keyboard, while the programming done mainly by voice with a support of standard input devices is titled "Method 2". As it comes to CER, the final score was 0,295 (30%) with a number of incorrectly interpreted commands being 67 and the total number of spoken commands was 227.

6.3 Discussion

The discussion about the results has been divided into three sections, focusing on different aspects: programming time, interaction with mouse and keyboard and CER. The results show that the time needed to complete a program via voice input is indeed longer, nonetheless it requires less interaction with input devices. The number of incorrect commands was relatively small, however, there is still a lot of room for improvement. More details can be read below.

6.3.1 Programming time

As it can be seen, completion of the test using Method 1 took only 5 minutes and 57 seconds, while Method 2 needed 24 minutes and 54 seconds. This in fact means that programming using CodeSpeech is around four times slower. That is a significant difference and it proves that CodeSpeech is not suitable to replace the standard method just yet. It is assumed that there are several reasons that make standard way faster. The first could be many of the options built-into IDE that speed up programming experience, such as auto-completion of text, keyboard shortcuts, problem solving suggestions, creation wizards and others. The second reason that can accelerate writing code is possibility of copying specific structures and refactoring them instead of typing them from the very beginning every time. Third reason that is suspected to have a relevance is error correction. Speech recognition systems, when recognized correctly do not introduce typos as it happens when typing. However, when recognition is not correct, trying to fix it might take longer than correcting a typo, especially when the result of recognition happens to be false more than once. Couple of times during the test a situation occurred, where one command had to be given many times over. The worst situation that has happened was in experimental free speech mode, when incorrect text was entered and the command to reverse last change was also falsely recognized. That resulted in appending incorrect text to the previous one rather than correcting an error. Now the reverse operation had to be done twice, and it could again be falsely recognized and so on. Finally, the author, who performed the test has over 5 years experience in programming, and even more in interacting with a computer using a mouse and keyboard while programming by voice is a new concept. Lots of mistakes were made, such as saying an incorrect command, stopping in the middle of the utterance to think about what comes next and forgetting about going back from free speech mode.

6.3.2 Interaction with mouse and keyboard

Although the time required to finish the program was longer in case of Method 2, the time of mouse and keyboard usage was shorter by 3 minutes 11 seconds decreasing it to 37,6%. Distance traveled by the cursor was shortened to 17,4%. Method 2 required 614 less keystrokes (29,9%) and the number of clicks and double clicks was decreased to 20% and 5% respectively. Mouse wheel was not used at all during the test.
It is clear that the usage of mouse and keyboard has been decreased quite substantially in programming by voice approach. One might argue that despite that, the increase of time needed to complete a simple program is not worth the change. The author of the thesis himself agrees with that statement, however, it is worth to remind that these are the results of the initial test and the system under evaluation is still a prototype. With the enhancement of the project and increase in functionality, usability and with better error correction a day might come when the tool becomes competitive in performance while keeping its benefits of limiting mouse and keyboard use.

6.3.3 CER

The performance of the system could be improved by decreasing CER, on which two factors have an impact, namely accuracy of speech recognition and strictness of interpretation based on grammar. Obviously, the better the recognition, the easier is to match the grammar, however it cannot be assumed that SR will always give perfect results, that is why handling it properly by interpreter should allow some flexibility to make up for mistakes done by SR. Improving both factors will result in smaller amount of needed repetitions an error corrections and and thus, shorter time of programming. In the meantime, the next section presents some ideas about how to possibly improve the system.

6.4 Ideas for improvement

During the evaluation a couple of different ideas of how to make the program better came up. In this section some of them are presented. At the beginning the focus of the discussion is on the ideas how to make recognition better as it is obviously one of the most important factors. This at times intertwines with the ideas for improvement of interpretation. Later the section pivots to discuss how to improve usability, and finally gives examples of features that increase performance of the system.

6.4.1 Interpretation of more propositions

Speech recognition systems solve classification problem. The results that are provided are based on probability. Many SR toolkits give the whole list of potential solutions, and it is not different in the case of Google API. The output returned by the API is a list of phrases, ordered by measure of confidence. It might happen, that the correct phrase is not on the surface but rather is somewhere deeper in the list. In the current version, CodeSpeech trustfully takes the top item and do not care about the others. Perhaps interpretation could be done on all of the propositions until a correct one is found? This of course could slow down the whole cycle, unless interpretation on different phrases is done in parallel.
An alternative to this approach is to add an additional feature, which would allow the user to iterate through the list of propositions, e.g. by running the command “try next”. Therefore every correct recognition done at the first try would not slow down the system, and in case of incorrect one the user would have control.

6.4.2 Context recognition

Another idea focuses on adding context to the process of recognition. Google Cloud Speech-to-Text is good in recognizing continuous speech, but that does not ensure correct results in systems with limited amount of words or phrases. Programming languages themselves have restricted list of keywords. It comes to mind, that a smart idea would be to provide some context in order to increase performance and omit cases in which the phrase like “change return type...” is being recognized as “Chun’s restaurant...”, which of course in the programming world does not make any sense (unless Chun’s restaurant is supposed to be used as a variable name). Since in CodeSpeech interpretation of recognized phrase is based on grammar, why not to use it as a context for recognition as well? Google API does not supply such an option, but CMUSphinx tools do. The grammar it is using is different from the one used by ANTLR, so its counterpart in the form of Java Speech Grammar Format (JSGF) would need to be created. That obviously comes with some consequences, namely whenever a change is to be introduced to the grammar, both files have to be modified accordingly. Perhaps this process could be automated.

Next concept of context based recognition relies on the third option provided by CMUSphinx, which is keyphrase recognition. Once a text file with specified phrases is created, it can then be used to trigger an action on a recognized keyphrase or single keyword ignoring the rest. That seems like a better solution in comparison to grammar, because grammar tend to be very strict, and if at least one unexpected word (such as an article) appears in the middle of a sentence, the match is not found. That makes keyphrase approach more flexible. It does sound promising, however, how it will work in practice must be investigated.

It may be that combining one of the aforementioned approaches (or both) with continuous, free speech recognition can give even better results. CMUSphinx allows for changing between modes during run time. That could require modifying the way voice commands are given to consist of phases. Initially, the SR is set up to keyphrase detection. Once the keyphrase has been detected the mode is switched to listening for continuous speech (when the name is to be spoken), and then back. An example of recognition divided into phases could look like this:

1. command “create new public method named” is given,
2. keyphrase is detected, and the mode switches to free speech listening,
3. when the user is notified, “get current state” name is spoken,
4. operation creating new method of name “getCurrentState” is created and the mode switches back to keyphrase recognition.

This way seems like it could again slow down the whole process a bit, but perhaps if the outcome will mean more accurate recognition, thus a decrease in the amount of repetitions needed, in the end it might turn out to actually be faster. In order to easier operate with names that already exist in the project, perhaps a good idea is to dynamically store them in the form of separate keyphrase file. This way at the beginning of name giving phase, the recognition could be performed in keyphrase mode (just the file is changed to the one consisting of names). If relatively strong match is found, the name will be brought up, but if not then probably a new name is being given, so recognition proceeds in continuous mode. This approach could increase precision and maybe decrease error rate in code reuse.

6.4.3 Combining recognition systems

In previous subsections the usage of CMUSphinx was considered to be applied in potential future improvement work, despite the fact that in the previous sections thi technology was rejected for not being good enough. It is true that CMUSphinx’s default models do not ensure a good recognition for non-native speakers. Even recommended acoustic adaptation did not provide enough improvement (would require much more adaptation than it has been done). It comes, however, with the tools to build completely new acoustic and language models based on a phonetic dictionary, which can be modified according to needs. Perhaps CMUSphinx will never be as good in continuous recognition as its commercial counterpart belonging to Google, nonetheless, it might prove good enough for the recognition of limited list of commands/words if the models are being built specifically for this purpose. Therefore the recognition of commands could be left to CMUSphinx in its more suitable modes mentioned earlier, while continuous recognition used e.g. for names could be left to Google Cloud Speech-to-Text or yet another tool. The process of building all models from scratch is going to be time consuming and will require a lot of work, but it may be worth considering anyway.

6.4.4 Better error correction

For now the only means of error correction works for the text editor, where source code is written. By saying “undo last” the last change in the text is reversed. In some situations, however, it might be necessary to perform reverse action more than once. In cases like this the command “undo last” needs to be spoken repeated couple of times. It gets even worse when this command is falsely classified and an extra text is entered. That adds to the stack of needed corrections. To make it better, it should be possible to give a number of times the reverse operation has to be made, for example “undo last three times”. That could speed up the process.

Additionally other means of error correction should be implemented, such as a possibility to change the wrong name, move a structure and basically to provide more options of manipulation, as well as allowing to use some of the IDE's proposed solutions in an easy way. This should be done not only for the text editor, but also for other elements, such as package explorer.

In SR tools such as Windows Speech Recognition, when the users are not satisfied with the inserted word, they can give a command "correct' and are then presented with a list of options that can be selected by saying a number of a line with correct phrase. Borrowing from that a similar list could be made available to the user when an incorrect operation was performed or when it is not clear what the user is trying to do. Perhaps this solution could also be used for future reference and help the system learn to avoid similar mistakes in the future.

6.4.5 Adaptation

Previous subsections inspired another idea. Some speech recognition systems allow for adaptation of the model with the recordings of the users to improve the results. Adding this as a feature could make sure that CodeSpeech achieves the same performance when working in different environments, voices, accents once it learns from those recordings. Whenever the users are not satisfied with the results they can start teaching mode in which they will have to read out loud a couple of sentences or perhaps specific commands. The process can be repeated a couple of times and with each improvement should be visible.

6.4.6 Feedback

Until now the focus was put on recognition, interpretation and operation execution, to make sure the system works. Currently there exists no feedback system built into the plugin, which makes it difficult to use. Whenever an error occurs or an exception is thrown, the user is not notified about that fact and therefore is unaware of why no action was performed or what he/she did wrong. Even if no error occurred, but simply the recognition was incorrect and nothing happens the user does not know the reason for it. A way to give back information of the current status should be implemented. Feedback system could be a text window (similar to the console) that displays either the recognized text, some details about performed action (if any) or simply an information that the system did not understand or wrong action was undertaken.

6.4.7 Improve voice control experience

Ideally, the end product's input is going to be flexible enough to invoke one operation with different sentences that have the same meaning in English. If this goal turns out to be hard to reach, maybe it can be worth to consider giving the users the possibility to define their own commands for each action. All keywords could be presented in table form in which respective words could be overwritten. After the modification the program reacts to different phrases. Despite which option will end up being in the finished product, it should be possible at any time for the user to open a list of phrases that are available in the current context. Even the author had to go back to the grammar file in order to check the proper way of saying a specific phrase.

Another observation made during the test was that whenever a break was made in the middle of the recognition e.g. between words or to think about the proper name, the sentence was broken in half and whatever has been said until this point was sent to recognition. That enforces on user to think first about what is to be said and then say it without a pause. That does not seem very convenient. A better approach would be to save the state of the current command wherever it was broken. This could be achieved by introducing more listeners (perhaps each for each state) and replace the current one with the new one every time a keyword is recognized. The context is passed in between the listeners anyway, so no information would be lost. When the users find themselves in the middle of the command which they do not want to finish, they could simply go back to the initial state by saying “cancel”. The current implementation surely allows for such a solution.

6.4.8 More features

The amount of features available in CodeSpeech is very limited. There are plenty of structures that cannot be created, modification possibilities are almost not existing and there are lots of options that IDE provide that are not accessible via CodeSpeech. The base of the system is more or less finished and it can be easily extended to provide new functionalities. However, adding new options takes time and so bringing the project to truly satisfying state will require large amounts of work.
With more options added the usability of the project would of course increase. Some examples of features to be implemented are:

- better editor navigation,
- text editing similar to this available in SR tools,
- possibility to select, copy and paste parts of code, project elements or AST nodes,
- possibility to call IDE commands such as indent of the code, performing a quick fix action, auto import of libraries etc.

Chapter 7

Conclusions

This chapter starts with a brief reminder of what was the reason behind launching the project in the first place. Later on, it summarizes what has been done until now, describes shortly the different phases of the project development, what is the current status, how well does it satisfy previously specified requirements and if it indeed fulfills its purpose and solves the problems it was created to solve. In the end there is a discussion about potential future work derived from the observations made during the whole process.

7.1 Why programming by voice

There are two main reasons behind the idea of programming by voice. First and foremost are repetitively reported musculoskeletal pains developed by long-term computer users, the author of the thesis being one of them. Some of the issues are suspected to be caused by repetitive movements and are therefore called "Repetitive Strain Injuries". There are studies confirming that the pain in upper extremities can be caused by excessive use of mouse and keyboard. If that is so, in order to solve this problem the time of usage of these devices should be limited. Another case where programming by voice could become useful deals with a special social group, namely people whose physical impairments do not allow them to use the mouse and keyboard but who are capable of speech. A tool enabling them to use their voices to write programs could help them to take upon an occupation of a programmer. This would supply the ever growing needs of this profession with more experts and at the same time help those people in their lives.

7.2 Summary of the development process

Once the idea came up a way of realization it to life had to be found. After doing some research and reading about similar projects, an initial concept of the project was made. It was decided that the program will be a part of an already existing programming environment, that it will integrate an already existing speech recognition tool, will be developed for Java programming language and be controlled by natural English sentences (also those spoken by a non-native speakers).

7.2.1 Tools selection

Eclipse IDE was selected as a programming and target platform almost from the very beginning. Main reason being the fact that the author is familiar with this environment, as well as that it can be easily extendable via plugins and it provides a whole toolkit for the development thereof with an extensive documentation.

The next step was to select speech recognition system, preferably open-source and free of charge. After small research and without getting into details CMUSphinx was decided upon. The system sounded promising and therefore its Java implementation - Sphinx4, was tested out. It turned out not to work so well, and so another of the CMUSphinx tools - Pocketsphinx was given a try, mainly for its keyphrase based recognition. Unfortunately, Pocketsphinx was not easy to apply into Java project and a lot of time was spent for the sole purpose of building the libraries just to be able to use it, and in the end to find out that even in keyphrase mode the speech recognition for non-native speaker is not of satisfying level. In order to improve the results, at first an acoustic model adaptation was performed, then also a new language model was build based on a phonetic dictionary, all in accordance to the instructions contained in the official documentation. Unfortunately, nothing brought expected results and therefore a search for another speech recognition tool was started. As another SR tool to be used, Google Cloud Speech-to-Text service was given a try and it turned out to be a good choice. The results of its recognition allowed for the project to resume its development. It was easy in integration and usage.
The time spent on CMUSphinx toolkit could have been spent on implementing more functionality into CodeSpeech project, nonetheless, it is not being considered wasted. The author has learned a lot during that process. A major take away from this experience is to try out different systems before deciding on one, and base the decision on not only the performance, but also on complexity and time needed for integration.

After the SR API was ready, a decision had to be made on how the voice control of the system should work. The initial idea was to use natural English sentences over command based approach. The reason is obvious, prior to operating with special commands, a learning process is required while in the ideal case a flexible solution could possibly allow to skip this process and let the user trust their intuition to control the program. Unfortunately, there is no unified way of reading code and every person does it in their own manner. Because of that, in order to be able to create a system with natural and intuitive control, a user study in this area has to be done first. To speed things up, the second solution based on command control was settled on. That led to an idea of utilizing grammar and together with it a parser program to interpret it. Commands defined by grammar turned out to allow for some flexibility, even more so with ANTLR being a parsing tool, as it allows for different implementations of so called (keyword) listeners that act differently on detected tokens throughout the process of traversing through a phrase. With more work and more options added, a system close to an ideal one might be possible to achieve. For now the expected input is rather strict, but still able to perform its task well enough for the first iteration.

7.2.2 Architecture design

Once all of the tools were ready, the time came to integrate them together to create a new product. The basic concept was already there, yet it was lacking necessary details. How will the different tools to communicate, what will be their relation etc. were the questions that had to be answered. Each tool was planned to be implemented separately with different responsibilities. Some of the answers came straight from the tools themselves. For example, the program was supposed to be an Eclipse plugin. That laid the foundations for the architecture. Basically what was needed to do was to follow an official documentation on plug-in development. It was then clear, that the main structure will be a class that is an instance of a plug-in itself. This instance was decided to be a "headquarters" for the rest of the tools, over which it had control.
Workflow between different parts was more or less predetermined, as at first speech has to be recognized and only then could be parsed and acted upon. However, instead of the result of recognition being directly passed to another tool for interpretation, it was decided to make it go through the plugin instance. This is to decrease coupling and give more control to the main class. Communication between different parts of the project is done via event system. Whenever an action that triggers a specific event happens, all of the registered listeners receive a notification with important data. The main listener of both, recognition and interpretation is the plugin class.

The hardest part was to come up with a way to go from interpretation to performing a task. Both of these processes are related, because action to be performed depends directly on the interpretation of the command. This relation and complexity led to constant changes during the development process.
The main problem was the sequence of words in natural English sentences on which commands are based versus the sequence of performing a task in programming. For easier understanding of the issue an explanation is given on an example. Usually, the process of performing a task is in this order: an object of a specific **type** and **name** is created, then all of its **attributes** are set and only when the object is ready it is used to perform a certain **task**. Now, given a phrase "create public static method main" it can be seen, that the verb determining the **task** to be performed comes first (create), right after that there are **attributes** (public, static), then there is a word defining the **type** to be created (method) and finally there is a **name** (main). The spoken English sentence is in reversed order.

Finally, understanding that the troublesome order of keywords and the sequential nature of the parser make it difficult to instantaneously perform correct operation, the full advantage of ANTLR's flexibility was taken and a build up approach was introduced. Because each of the words might have a different interpretation depending on the context it was decided to use different listeners. The first verb determines which listener will be used in parsing. Then throughout the whole process an interpreter context object gathers all of the necessary information such as attributes during the whole process of interpretation. Once a structure keyword is recognized it initializes a specific operation based on the type of a task and this structure. Structures are represented by models that consists of all attributes important for them.

Before settling for the operations and models, other ideas were tampered with. Initially, all operations were supposed to be performed by specific objects called managers.

The problem appeared then, how to group them. Should they be focused on operation, like a `CreationManager` responsible for creation of all of the structures, or should they be structure focused, e.g. `MethodManager` performing all operations on methods. What's with operations not involved with code? To perform operations on AST there would be an `ASTManager` and to perform operations on UI - `UIManager` (the last two actually made it to the final version, with some modifications). It was noticed then that the problem can be solved by implementing the "double dispatch" through the Visitor pattern. Performed operation would depend on the types of two instances: operation and a model. During the process of parsing the command, proper model and operation objects would be created, such as `CreateOperation` and `VariableModel`. Then a proper overloaded method of `perform` of `CreateOperation` taking `VariableModel` as parameter would be invoked. After some time the program grew, and with it more and more models were added, resulting in many inevitable changes in operation classes, ending with few of them having empty not implemented methods, and others not being dependent on models at all. That led a will to simplify the structure, keeping the concept of operations and models but resigning from the Visitor pattern. Right now each operation class performs only a single action taking a single object passed as a parameter. The object has to be of proper type but that is being handled inside its method. This results in compact, but many operation classes. Whether this is a good structure or not is debatable, the author himself is not certain of that. Nevertheless, it turned out that management of such classes is easy (unless the abstraction needs to be changed, following by the need of changing every single realization) and the addition of new functionalities is as simple.

7.2.3 Implementation

Design and implementation phases were constantly being changed. At first, a basic architecture was designed, then when it came to the implementation some new ideas came up, and so the design was modified. An example of this process was presented in the previous subsection. A slight different, but worth mentioning experience that the author had was introduction of abstractions. The search for a satisfying SR API resulted in having three implementations of different technologies that were supposed to do the same. Obviously a need for unification appeared. That lead to creation of an abstraction for speech recognition, with each SR implemented so far being a realization of it. Combining this with a Factory design pattern, it not only made it possible to easily switch between different APIs, but also to extend the project by new ones. This experience can be a real life example of the advantage that working with abstractions bring. Of course sometimes it is not easy to design such a system at the very beginning, in the first place the designer has to be familiar with the general idea of how the system should work. This is what has happened in this case. Working with different APIs at the beginning helped the author to gain an understanding of how the technology works and therefore made it easier to derive an abstraction.

This iterative approach was a repeating pattern throughout the whole creation process in a fashion of agile development, with each iteration going back to the previous phases if necessary.

7.3 Conclusions

The final product satisfies most of the requirements at least in partially, but unfortunately not all of them due to time limitations. From the functional requirements, what is lacking is an option to run and debug the program. Nevertheless, extending the project by these two features should be quite easily doable. The rest of the functionality is implemented, at least in a minimal form. Of course the terms used in the requirements are quite broad, because under the phrase “create source code” lie many features, such as creation of variables, methods etc., which could be a separate requirement on its own. For the purpose of simplification those specific requirements were generalized and put into one category.

Non-functional requirements are also not entirely fulfilled. For example, the program is easy to use, however it still requires a bit of learning due to the strict commands. It does recognize English sentences, even those spoken by non-native speakers. The system is rather reliable, meaning that it does not lose data and does not crash. There are times when exceptions are thrown, but they are being caught and thus the usage of the program can be continued. In some cases it is possible to correct mistakes, but this feature still could be improved. If the program is intuitive is unknown and to check that a user study has to be performed. The system does not inform about the errors unfortunately, or gives any feedback for that matter (except when the operation is performed). This was broadly discussed in one of the previous subsections in potential extension by feedback system. The response of the project is relatively fast, but is not immediate and that is due to the recognition process which in average takes around 2 seconds. The system was tested under Windows 10 and because the program is an Eclipse plugin and Eclipse is available for Linux it is assumed that it will work there as well, however, this is yet to be tested.

In the end a prototype of a programming by voice tool was created. At first the functional and non-functional requirements were defined together with planned use cases. Then, an initial structure was designed and in iterative process of agile development the first version of the tool was finished. After that, an evaluation of the system was done by the author. The arising results showed that in the current state the prototype is not ready to completely replace the standard way of programming through the use of mouse and keyboard, but it can decrease the amount of interaction with these devices and thus possibly minimize the risk of RSI occurrence. It is worth to remind that the program under test is still just a prototype and has plenty of room for improvement. It is assumed that once it is enhanced and it provides more features it could become a substitution for mouse and keyboard in the future.

7.4 Future work

From this point on there are a couple of things that need to be done. The first one is to of course enhance the project further following some of the ideas given in the previous chapter. During the evaluation a realization was made, of which features should be implemented first in order to speed up the increase of project’s performance. Ideally, the second iteration should already allow creation of a complete program done entirely by voice, without the need of mouse and keyboard.

In parallel to the development of the second iteration, a user study should be performed in order to answer the question of how the voice input should look like according to the programmers. The research could take the form of Wizard of Oz study, in which the users sit in front of a workstation. They do not need to interact with the workstation manually at all. In order to perform an action the users speak out loud what action they want to perform and these actions are then performed by another person at another computer which shares the screen to the test station. All changes are being displayed to the user in real time. This way the subjects under study will have a real feeling of operating by voice. Phrases spoken by them will be recorded and analyzed. This study will help to form a grammar in such a way to allow many different possibilities of performing the same task using different sentences, thus making program operation feels more natural and therefore decreasing the time needed for learning, ideally to zero.

Once the second iteration is ready and the results of the user study are applied to it, another evaluation of the project should be done. This time it would involve more people, ideally around twenty. All of the participants should be familiar with the concept of Object Oriented Programming and Java programming language, preferably with an Eclipse IDE as well. In this study the users will have to write the same program using standard method and voice (similarly to the way it has been done in a test described in this thesis). In order to nullify learning effect of the program, the participants will be divided into two groups. The first group will begin with coding in a standard manner and the second one, with voice. If needed, a list of possible commands can be presented to study subjects either before the start or can be made available for the whole time (perhaps implemented into the system as an extra feature). Both methods will be then compared as to the time of completion. In addition, CER will be independently measured. After the study a questionnaire will be provided to the participants in order to gather information of subjective feelings about the system, such as if it is easy, intuitive, frustrating and if they would be willing to use it in the future.

Next user study will determine how the learning factor affects the performance. The idea as of now is to repeat the user study ten times over with the same study group but with different example programs. Each time will be recorded and the results will be compared. This will show if there is a potential for programming by voice to increase the performance to such an extent that it can be competitive to the standard way, or perhaps even better.

Changing the way of interaction with the computer might influence programming in a great manner. In case voice coding becomes the main method of writing programs, it might not be feasible to do so in a form of text. Using specified commands and manipulating Abstract Syntax Tree might be a better solution. As an effect this could also have consequences in the representation of the program. When the text files is no more needed, perhaps some sort of graphical representation would become a natural replacement. This is an interesting topic for Human-Computer Interaction research and once everything else is done and the program is completed, a window will open for scientific studies in this field.

Appendix A

CD-ROM/DVD Contents

- **at.ooe.fh.mc.codespeech project** - folder containing libraries, source code and other necessary project files
- **codespeech plugin** - folder containing plugin *jar* file nested in *plugins* folder. In order to use the plugin in an Eclipse instance, this folder has to be placed in *dropins* folder in the IDE's root directory
- **related work** - folder containing some of the related work used for this thesis
- **CodeSpeech.pdf** - Master Thesis in *pdf* file
- **thesis latex** - folder containing LaTeX sources

Appendix B

Source Code

Program B.1: Soundable interface.

```java
package animals.interfaces;

public interface Soundable {

  public String speak();
  public String sound();
}
```

Program B.2: Animal abstract class implementing Soundable interface.

```java
package animals.models;

import animals.interfaces.Soundable;

public abstract class Animal implements Soundable {

  String name;

  public Animal(String name) {
    this.name = name;
  }

  public String speak() {
    return name + " says " + sound();
  }

}
```

Program B.3: Cow class extending Animal.

```java
package animals.models;

public class Cow extends Animal {

  public Cow(String name) {
    super(name);
  }

  @Override
  public String sound() {
    return "mooo";
  }

}
```

Program B.4: Horse class extending Animal.

```java
package animals.models;

public class Horse extends Animal {

  public Horse(String name) {
    super(name);
  }

  @Override
  public String sound() {
    return "niaagh";
  }

}
```

Program B.5: Sheep class extending Animal.

```java
package animals.models;

public class Sheep extends Animal {

  public Sheep(String name) {
    super(name);
  }

  public String sound() {
    return "beee";
  }
}
```

Program B.6: Animals class with main method printing to the console.

```java
package animals;

import animals.models.Animal;
import animals.models.Cow;
import animals.models.Horse;
import animals.models.Sheep;

public class Animals {

  public static void main(String[] args) {
    String horseName = "Unicorn";
    String cowName = "Bully";
    String sheepName = "Little Lamb";

    Animal horse = new Horse(horseName);
    Animal cow = new Cow(cowName);
    Animal sheep = new Sheep(sheepName);

    System.out.println(horse.speak());
    System.out.println(cow.speak());
    System.out.println(sheep.speak());
  }

}
```

References

Literature

[1] Stephen C. Arnold, Leo Mark, and John Goldthwaite. “Programming By Voice, Vocal Programming”. In: *Proceedings of the fourth international ACM conference on Assistive technologies.* ACM. 2000, pp. 149–155 (cit. on pp. 6, 11).

[2] Andrew Begel. “Programming by voice: A domain-specific application of speech recognition”. In: *AVIOS speech technology symposium–SpeechTek West.* Citeseer. 2005 (cit. on pp. 7, 11).

[3] Andrew Begel. “Spoken language support for software development”. In: *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on.* IEEE. 2004, pp. 271–272 (cit. on p. 7).

[4] Claudio Bettini and S Chin. “Towards a speech oriented programming environment”. In: *Computer and Communication Systems, 1990. IEEE TENCON’90., 1990 IEEE Region 10 Conference on.* IEEE. 1990, pp. 592–595 (cit. on p. 10).

[5] Marat Boshernitsan. *Harmonia: A flexible framework for constructing interactive language-based programming tools.* Computer Science Division, University of California, 2001 (cit. on p. 7).

[6] Leana Morgan Bouse. “Voice and gesture development environment: keyboard free programming”. PhD thesis. Texas A&M University, 2017 (cit. on p. 9).

[7] Lars Peter Brandt et al. “Neck and shoulder symptoms and disorders among Danish computer workers.” *Scandinavian journal of work, environment & health* 30.5 (2004), pp. 399–409 (cit. on p. 2).

[8] Che-hsu Chang et al. “Daily computer usage correlated with undergraduate students’ musculoskeletal symptoms”. *American journal of industrial medicine* 50.6 (2007), pp. 481–488 (cit. on p. 2).

[9] Alain Desilets. “VoiceGrip: a tool for programming-by-voice”. *International Journal of Speech Technology* 4.2 (2001), pp. 103–116 (cit. on pp. 5, 11).

[10] Alain Désilets, David C. Fox, and Stuart Norton. “Voicecode: An innovative speech interface for programming-by-voice”. In: *CHI’06 Extended Abstracts on Human Factors in Computing Systems.* ACM. 2006, pp. 239–242 (cit. on p. 5).

[11] Felienne Hermans, Alaaeddin Swidan, and Efthimia Aivaloglou. “Code Phonology: an exploration into the vocalization of code”. In: *Proceedings of the 26th Conference on Program Comprehension.* ACM. 2018, pp. 308–311 (cit. on p. 11).

[12] MD Islam, Hosne Mobarak, MD Islam, et al. “Voice command based android java code generator”. PhD thesis. BRAC University, 2018 (cit. on p. 9).

[13] Christina Funch Lassen et al. “Elbow and wrist/hand symptoms among 6,943 computer operators: A 1-year follow-up study (the NUDATA study)”. *American journal of industrial medicine* 46.5 (2004), pp. 521–533 (cit. on p. 2).

[14] Christina F Lassen et al. “Risk factors for persistent elbow, forearm and hand pain among computer workers”. *Scandinavian journal of work, environment & health* 31.2 (2005), pp. 122–131 (cit. on p. 2).

[15] Jean K Rodriguez-Cartagena et al. “The implementation of a vocabulary and grammar for an open-source speech-recognition programming platform”. In: *Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility*. ACM. 2015, pp. 447–448 (cit. on p. 11).

[16] Lucas Rosenblatt et al. “Vocal Programming for People with Upper-Body Motor Impairments”. In: *Proceedings of the Internet of Accessible Things*. ACM. 2018, p. 30 (cit. on p. 6).

[17] Eric B Schlossberg et al. “Upper extremity pain and computer use among engineering graduate students”. *American journal of industrial medicine* 46.3 (2004), pp. 297–303 (cit. on p. 1).

[18] Archana R Shinde. “Natural Language Interface for Java Programming: Survey”. *International Journal on Recent and Innovation Trends in Computing and Communication* 5.11 (2017), pp. 17–20 (cit. on p. 8).

[19] Lindsey Snell and Mr Jim Cunningham. “An investigation into programming by voice and development of a toolkit for writing voice-controlled applications”. *M. Eng. Report, Imperial College of Science, Technology and Medicine, London* (2000) (cit. on p. 8).

[20] Mark Stephen Tremblay et al. “Physiological and health implications of a sedentary lifestyle”. *Applied physiology, nutrition, and metabolism* 35.6 (2010), pp. 725–740 (cit. on p. 2).

[21] Swenne G Van den Heuvel et al. “Loss of productivity due to neck/shoulder symptoms and hand/arm symptoms: results from the PROMO-study”. *Journal of occupational rehabilitation* 17.3 (2007), pp. 370–382 (cit. on p. 2).

[22] Maurits Van Tulder, Antti Malmivaara, and Bart Koes. “Repetitive strain injury”. *The Lancet* 369.9575 (2007), pp. 1815–1822 (cit. on p. 2).

[23] K Walker-Bone and Cyrus Cooper. “Hard work never hurt anyone: or did it? A review of occupational associations with soft tissue musculoskeletal disorders of the neck and upper limb”. *Annals of the rheumatic diseases* 64.10 (2005), pp. 1391–1396 (cit. on p. 2).

[24] Annalee Yassi. “Repetitive strain injuries”. *The Lancet* 349.9056 (1997), pp. 943–947 (cit. on p. 2).

Online sources

[25] *ANTLR*. URL: https://www.antlr.org/ (visited on 06/26/2019) (cit. on p. 24).

[26] *ANTLR documentation*. 2019. URL: https://github.com/antlr/antlr4/blob/master/doc/index.md (visited on 07/30/2019) (cit. on p. 26).

[27] *Building Rich Client Applications with Eclipse 4*. Jan. 2013. URL: https://www.youtube.com/watch?v=IbMVDxgLYdk (visited on 06/26/2019) (cit. on p. 14).

[28] *CMUSphinx Documentation*. URL: https://cmusphinx.github.io/wiki/ (visited on 06/26/2019) (cit. on p. 20).

[29] *Eclipse documentation*. 2019. URL: https://help.eclipse.org/2019-06/index.jsp (visited on 07/30/2019) (cit. on p. 18).

[30] *Google Speech-to-Text Pricing*. URL: https://cloud.google.com/speech-to-text/pricing (visited on 06/26/2019) (cit. on p. 23).

[31] *OSGi (Open Service Gateway Initiative)*. Mar. 2011. URL: https://searchnetworking.techtarget.com/definition/OSGi (visited on 06/26/2019) (cit. on p. 14).

[32] *VoiceCode*. Feb. 2007. URL: https://www.youtube.com/watch?v=A7f9Iik3q58 (visited on 06/28/2019) (cit. on p. 5).

[33] *Voicecode.io*. 2017. URL: https://voicecode.io/#about (visited on 07/17/2019) (cit. on p. 10).

Check Final Print Size

— Check final print size! —

width = 100mm
height = 50mm

— Remove this page after printing! —