
Oral Programming Interface
Using
Speech Recognition Technologies

By
TSAOUSIS APOSTOLOS



Department of Electrical & Computer Engineering
UNIVERSITY OF THESSALY

Supervised by,

Lalis Spyros, Associate Professor
Antonopoulos Christos, Assistant Professor

OCTOBER 2017

[This page was intentionally left blank]

DEDICATION

This thesis is dedicated to my family and especially to my mother, Annita.

She has always been there for me with her endless, unselfish, true and unconditional love and support.

She always encouraged me to follow my dreams, believe in myself and not let anything bring me down. Without her, I wouldn't be able to accomplish anything.

This dedication is a least token of gratitude and recognition of her self-sacrifices.

– I am so proud to be called your son

[This page was intentionally left blank]

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to my supervisors, Professors Spyros Lalis and Christos Antonopoulos, for their continuous support and guidance not only for the completion of my thesis, but also throughout my entire academic studies. Subconsciously, they acted as role models, by inspiring me with their passion for knowledge and education and consequently helped me develop my personal and programming skills. It has been an honour to be your student.

I would also like to thank all the department's faculty and staff with whom I had an excellent collaboration all of those years. Their contribution was significant as well.

Finally, last but not least, I want to thank my fellow students and friends who stood by my side in every way during the good and most importantly the bad moments and shared with me this unforgettable and exciting journey.

Ο σκοπός της εργασίας, η οποία παρουσιάζεται σε αυτή την διπλωματική διατριβή, είναι η ανάπτυξη μιας προφορικής διεπαφής προγραμματισμού με χρήση τεχνολογιών αναγνώρισης ομιλίας. Η μέθοδος αυτή αποτελεί μια ελκυστική εναλλακτική προσέγγιση έναντι της κλασικής μεθόδου της πληκτρολόγησης, ειδικά για προγραμματιστές με κάποια μορφή κινητικής αναπηρίας, οι οποίοι αντιμετωπίζουν δυσκολίες στην χρήση του πληκτρολογίου.

Ένα πρωτότυπο ενός λειτουργικού συστήματος συζητείται, το οποίο περιλαμβάνει μια mobile εφαρμογή για το λειτουργικό σύστημα Android και ένα plug-in για το ολοκληρωμένο περιβάλλον ανάπτυξης IntelliJ IDEA. Η mobile εφαρμογή χρησιμεύει ως διεπαφή χρήστη για την εισαγωγή εντολών με λεκτικό τρόπο. Μετατρέπει την καταγεγραμμένη ομιλία σε κείμενο χρησιμοποιώντας το API αναγνώρισης ομιλίας της Google και στέλνει το κείμενο στο plug-in. Με τη σειρά του, το plug-in αναλύει το ληφθέν κείμενο, το αντιστοιχίζει σε μία από τις υποστηριζόμενες εντολές και εφαρμόζει την εντολή στις εσωτερικές δομές δεδομένων του IntelliJ για να εκτελέσει την επιθυμητή ενέργεια, όπως το άνοιγμα ενός project ή ενός αρχείου προγράμματος και την προσθήκη ή την επεξεργασία του πηγαίου κώδικα.

Αρχικές, απλές δοκιμές που πραγματοποιήθηκαν με το πρωτότυπο για προσωπική χρήση, δείχνουν ότι η προτεινόμενη προσέγγιση έχει τη δυνατότητα να μειώσει σημαντικά το χρόνο που χρειάζεται για άτομα με κινητικές αναπηρίες να γράψουν κώδικα.

ABSTRACT

The objective of the work presented in this thesis is to develop an oral programming interface based on speech recognition technology. This method is an attractive alternative to typing, especially for programmers with physical disabilities, who have difficulty using a keyboard.

A working system prototype is discussed, which comprises a mobile application for the Android operating system and a plug-in for the integrated development environment of IntelliJ IDEA. The mobile application serves as the user interface for inputting commands in a verbal manner. It transforms the recorded speech to text, using the Google speech recognition API, and sends the text to the plug-in. In turn, the plug-in parses the text received, maps it to one of the supported commands, and applies the command to the internal data structures of IntelliJ in order to perform the desired action, such as opening a project or a program file and adding or editing source code.

First, simple tests that have been performed with the prototype for personal usage, show that the proposed approach has the potential of significantly reducing the time it takes for people with physical disabilities to write code.

TABLE OF CONTENTS

	Page
DEDICATION	i
ACKNOWLEDGEMENTS	ii
ΠΕΡΙΛΗΨΗ	iii
ABSTRACT	iv
TABLE OF CONTENTS	v
1. INTRODUCTION	1
1.1 Motivation	1
1.2 Objectives	2
1.3 Structure	2
2. SYSTEM OVERVIEW	4
3. ANDROID APP	6
3.1 Overview	6
3.2 Why Android?	6
3.3 System requirements	6
3.4 Codability's UI	7
3.5 Google Speech API	8
3.6 Discovering IntelliJ	9
4. INTELLIJ PLATFORM	11
4.1 Why IntelliJ?	11
4.2 Architecture	11
4.3 PSI	14
5. CODABILITY PLUG-IN	17
5.1 Overview	17
5.2 Web server	17
5.3 Command hierarchy & syntax	18
5.4 Command decoding	20
5.5 Command execution	22
5.6 How to extend the plug-in	24

6. EVALUATION	26
7. CONCLUSION	28
BIBLIOGRAPHY	29
APPENDIX A	30
A.1 Setting up the development environment	30
A.2 Codability's source code	32

[This page was intentionally left blank]

INTRODUCTION

1.1 Motivation

Nowadays, in the era where technology thrives, the demand for programmers and coding has grown beyond any expectation. Programmers spend most of their time coding in their computers. It is completely usual to write hundreds lines of code per day, which corresponds to thousands of keystrokes per day.

It has been recently discovered, that as a consequence of typing for long hours, a significant number of programmers suffer at some point in their career from repetitive strain injuries (RSI) ^[1]. RSI is a general term, used to describe the pain felt in muscles, nerves and tendons caused by repetitive movement and overuse.

Moreover, there are cases where a programmer faces a type of physical disability. For example, I myself am a quadriplegic, meaning that I have very limited control of my upper limbs.

Throughout my academic studies, I have worked on several projects and one of the main problems, that I have encountered was that my typing speed wasn't in the same level compared to that of my colleagues.

Therefore, I was thinking what could be done in order to make this everyday process easier, faster and less painful. The solution I came up with, was no other than voice.

[1] https://en.wikipedia.org/wiki/Repetitive_strain_injury , Repetitive strain injury definition by Wikipedia

Voice-based interaction is not a new concept. It is already used for the hands-free user interaction with mobile phones. It requires almost no effort and it is typically much faster than using a physical keyboard or a virtual keyboard via a touchscreen. So, why not start programming using speech, instead of using our hands?

1.2 Objectives

The purpose of this thesis is to demonstrate the concept of an oral programming interface using speech recognition technology. The final goal is to show that coding by speech can be achieved through a structured, well-defined interface and that this approach can provide a significant speedup in typing performance, especially for programmers with a form of a physical disability.

This is accomplished by developing a plug-in for the IntelliJ IDEA platform, referred to as Codability. Codability uses an Android App as an interface. The user talks directly to his smartphone, the speech is then being converted to text using the powerful Google Speech API, and the result is sent to the IntelliJ's plug-in which, in turn, parses the text according to a given syntax and generates the actual commands that are issued by IntelliJ.

Further details regarding the implementation, architecture and functionality of this plug-in will be analysed in the next chapters.

1.3 Structure

This thesis is divided into six main parts, followed by a conclusion.

Chapter 2 acts as an introductory for the rest of the chapters. It provides a high level overview of the system and briefly discusses the main steps of

the system's lifecycle to give an insight of what follows.

Chapter 3 discusses the implementation of the Android App. Initially it shows the UI of the App and provides a brief explanation of each component's functionality. Following, a more in-depth analysis of the speech technology used in this project, the Google's Speech API, takes place. Finally, the procedure used to discover the IntelliJ platform is shown.

Chapter 4 presents the IntelliJ IDEA platform. It provides the reader with the basic concepts of the system's architecture, how plug-ins are developed as components, specific thread invocation and multi-threading rules, information about the file system, and how IntelliJ parses files and creates the syntactic and semantic code model.

Chapter 5 discusses the implementation of the Codability plug-in. It presents how the plug-in receives the data sent by the mobile application, how this data is mapped into structured commands, and how these commands lead to the actual modification of the internal syntax tree of IntelliJ.

Chapter 6 provides a first evaluation of our prototype.

Finally, chapter 7 concludes this thesis and discusses possible improvements and extensions of the plug-in.

SYSTEM OVERVIEW

This chapter presents a high level abstraction of the system in order to provide a quick overview of the main components of the system that was developed in this thesis.

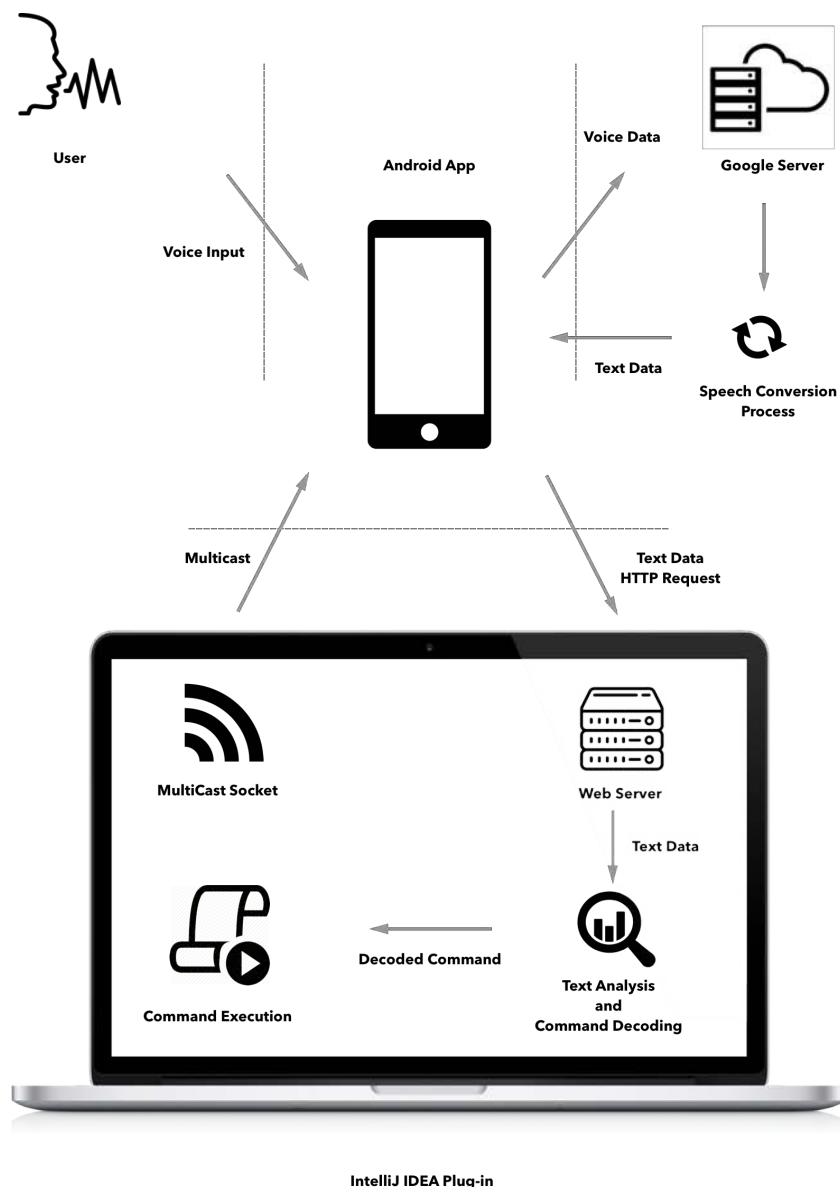


FIGURE 2.1 - SYSTEM OVERVIEW

As shown in Figure 2.1, the system consists of a mobile application and a plug-in for the IntelliJ platform. The interaction between those components is briefly discussed below.

As long as the plug-in is running, it periodically broadcasts to a pre-defined multicast address the connection details for IntelliJ's built-in web server, such as IP and port number. In turn, the mobile application listens to this multicast address to discover the contact information of the web server.

The user dictates commands orally to the mobile application. The voice data are sent to Google's server for speech recognition and conversion to text. The returned text is then sent to the plug-in.

When the plug-in receives the text, it performs an analysis based on well-defined syntax rules in order to parse and then decode the user command.

Finally, the decoded command is sent to the command execution unit, where a function that corresponds to the command is selected and executed.

3.1 Overview

The purpose of the mobile application is to act as an interface between the user and the IntelliJ IDE. The user dictates commands, then the application collects the speech input and sends this data via the Google's Speech API for conversion to text. To simplify the procedure and in order not to consume a lot of energy, no further processing is done by the mobile device. The mobile application receives the data from Google's servers and directly sends them to the plug-in for further processing.

3.2 Why Android?

Android was preferred over other OS solutions, primarily because of its open source nature, in conjunction with the fact that it is based on the familiar Java language. In addition, Android comes with a built-in API support for Google's speech recognition service, which is quite accurate, free, has no severe usage limitations and can also work in an offline mode on certain Android devices.

3.3 System requirements

The mobile application was developed for a Meizu M2 device, running on Android 5.1 Lollipop and is optimised for this particular setup. Some of the UI elements may not render properly on other devices due to different screen resolutions. Also, some features may not work on different versions of Android.

The table below summarizes the technical specifications of the Meizu M2.

Meizu M2	
OS	Android 5.1 Lollipop
CPU	Quad-Core 1.3 GHz Cortex-A53
RAM	2 GB
Resolution	720 x 1280 pixels

TABLE 3.1 MEIZU M2 TECHNICAL SPECS

An Internet connection is recommended, but not required in case the device supports offline speech recognition. Finally, in order to establish the communication channel between the mobile device and the PC running the IntelliJ, the mobile device must have WiFi capability and must be connected to the same WiFi network with the PC.

3.4 Codability's UI

As the main concept of this project is to have to as little hands-on interaction as possible with the system, user input is supported via a simple and user-friendly UI. It features two buttons and a text view.

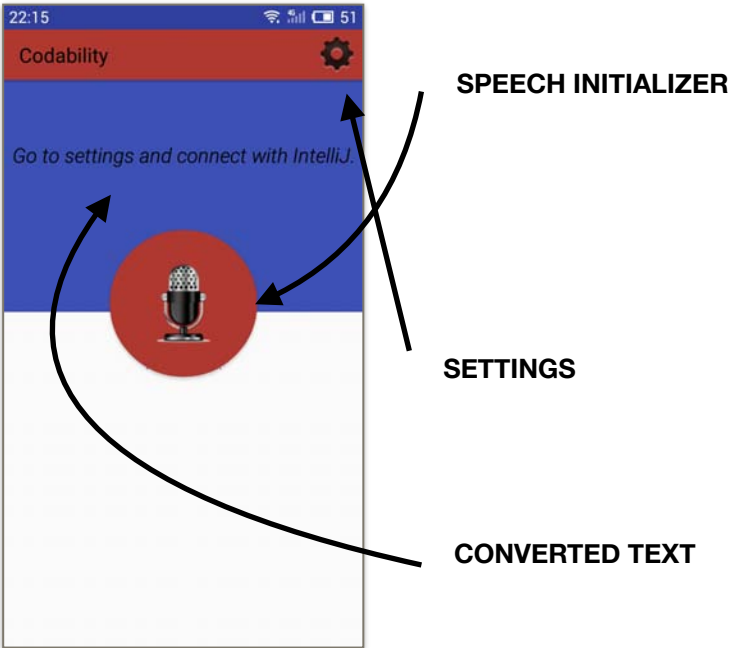


FIGURE 3.1 - STARTUP SCREEN

The main button (microphone icon) in the center of the screen is responsible for triggering the speech recognition process. The upper right button (gear icon) invokes the settings menu, from which the user can choose either the connect or the help option. Finally, the text view, above the microphone button inside the blue section, is used to display the commands that were recorded and converted to text by the Google Speech API.

3.5 Google Speech API

The speech recognition technology used in this project is Google Speech. Google Speech provides an API that enables developers to convert audio to text. It applies advanced deep learning neural network algorithms to the user's audio for speech recognition with increased accuracy.

Android doesn't provide continuous speech recognition by default. However, the target was for the user to be able to issue commands in a fluent non-disruptive way. To achieve this, the functionality of the `SpeechRecognizer` class was extended by implementing a new `RecognitionListener`. The `SpeechRecognizer` class provides access to the speech recognition service, whereas the `RecognitionListener` interface is used for implementing functions that are responsible for receiving notifications from the `SpeechRecognizer`, when recognition related events occur. Such events include `onResults()`, `onReadyForSpeech()`, etc .

This way, the user isn't prompted every time to press a button in order to initiate the speech recognition process for the next command. Instead, the installed `RecognitionListener` automatically re-activates voice recognition when the results of the previous speech recognition attempt return.

Figure 3.2, illustrates a flow diagram of this process. Initially, the speech recogniser listens for voice input. If no voice input is detected within a pre-defined time limit, then the speech recogniser is launched again. Otherwise, if the user's speech is recorded by the speech recognizer, then

the data are sent to Google. The recording process lasts until the user pauses for a few seconds. Everything being recorded in that session are being processed for conversion. On the retrieval of the results, an `onResults()` event is being triggered. In case the converted text corresponds to a stop command, the continuous speech recognition process is being terminated, else the control will return to the initial state.

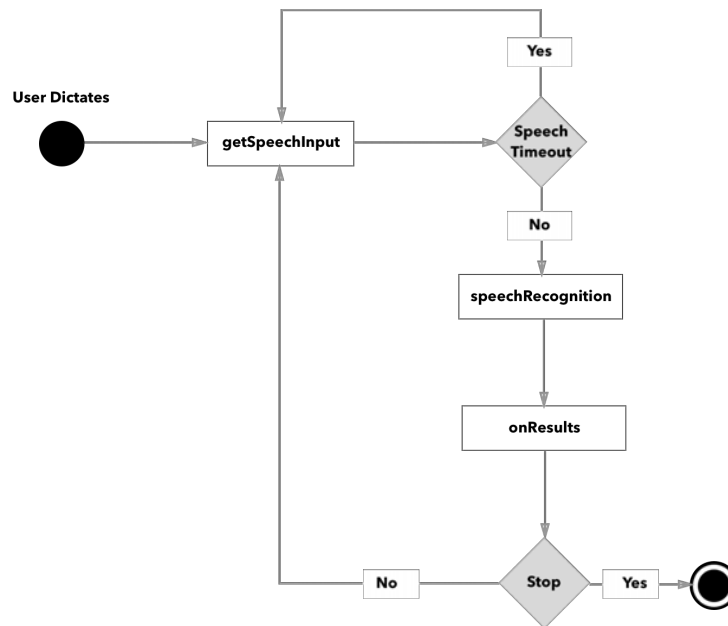


FIGURE 3.2 - CONTINUOUS SPEECH RECOGNITION FLOW DIAGRAM

3.6 Discovering IntelliJ

The mobile application needs to establish a connection with the IntelliJ plug-in, so that the plug-in can receive the text produced as a result of the speech recognition process. For this process the plug-in beacons the necessary contact information (IP address and port number).

On the mobile phone the user has to tap the Settings icon and then press the Connect option from the menu. By doing so, the mobile application joins and starts listening on the multicast address where the plug-in

broadcasts its beacon. Eventually, a beacon with the contact details of the IntelliJ's plug-in will be received, and the mobile application will be able to open a connection with it. The user is notified by a pop-up window that the mobile application successfully connected with the plug-in and his is prompted to tap on the microphone icon to start dictating. Figure 3.3 shows this process.

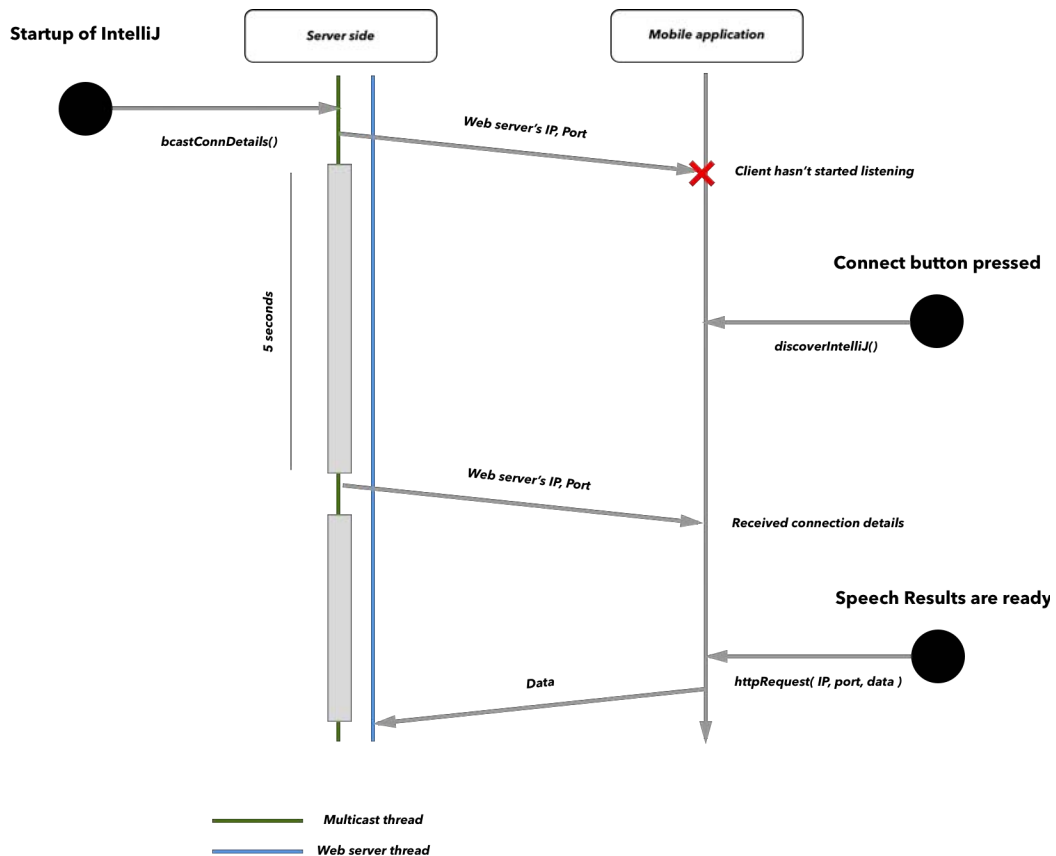


FIGURE 3.3 - DISCOVERY SERVICE

4.1 Why IntelliJ?

IntelliJ IDEA is a Java integrated development environment (IDE) for developing computer software. It is one of the most popular IDEs used amongst programmers. One of the main reasons for that, is that it offers an open-source IDE, the IntelliJ IDEA Community Edition, which is free to download and customise or extend according to the user's needs.

For the purpose of this work, IntelliJ was a particularly suitable platform because of its inherent extensibility support through plug-ins. For the development of those plug-ins IntelliJ provides a powerful SDK ^[1]. Furthermore, very important and helpful is the community supporting the IntelliJ IDEA Open API and Plugin Development forum.

4.2 Architecture

Component Model

IntelliJ is very modular in its nature, essentially being a collection of plug-ins (components) that are connected together to provide the desired functionality. Everything inside the IntelliJ platform, for example the menu, the editor or the project manager, is a component.

There are three levels of components, depending on the activity and scope of each plug-in. Application-level components have a global scope and are initialised when the IDE starts. Project-level components, on the other hand,

^[1] Unfortunately, the API is rather weakly documented, despite ongoing efforts to fix this issue.

are instantiated by the IDE for every project instance. Accordingly, module-level components are instantiated for every project's module loaded.

Filesystem

The IntelliJ platform introduces several concepts to handle file-related operations.

- Virtual files: IntelliJ offers a VFS (Virtual File System) for representing files on a file system. A virtual file corresponds to an actual file in the local file system. The VFS level deals only with binary content. The contents of a virtual file are accessed as a stream of bytes, while higher-level semantics like encodings and line separators are handled by higher system layers. The plug-in SDK doesn't provide support for implicitly creating virtual files programmatically.
- Documents: Documents represent the contents of virtual files. A document is an editable sequence of Unicode characters. Documents are volatile objects, that are dynamically created when the contents of a virtual file are accessed, but as opposed to virtual files which are persistent, they are automatically garbage-collected if not referenced. Unlike virtual files, new documents can be explicitly created programmatically.
- Program Structure Interface (PSI) files: A PSI file is the root of a structure that represents the contents of a file in the form of a hierarchical tree structure of PSI elements in a particular programming language. There are PSI implementations for various programming languages, such as PsiJavaFile (Java), XmlFile (XML), PyFile (Python) and more. Unlike virtual files and documents, which are application scoped, PSI files are project scoped. This way, each project can work on its own PSI instance for a file, which may be shared among projects.

Threading Rules

User interaction in IntelliJ is performed from within a specific, so-called UI thread. This runs from within the window manager in order to collect input from the user and to present output to the user. As a general rule, to keep an application responsive, the UI thread shouldn't be blocked by performing time-consuming operations.

Ideally, anything that's not directly related to user interaction shouldn't be performed in the UI thread. To this end, one may create additional threads that run concurrently to the UI thread and can perform tasks in the background. However, proper synchronization is needed when threads access shared state and data structures.

For this purpose, IntelliJ employs a single reader/writer lock for all code-related data structures, such as the PSI and the VFS. This lock is implicitly accessed through corresponding methods of the `ApplicationManager` service. `ApplicationManager` grants access to IntelliJ's core application-wide functionality and methods. It supports two main types of actions which can access the PSI and other IDEA data structures: read actions (which do not modify the data) and write actions (which modify some data). Reading is allowed from any thread, but threads other than the UI thread must explicitly synchronize by wrapping such operations using the `runReadAction()` method of the `ApplicationManager`. Writing is only allowed from the UI thread, and write operations always need to be wrapped in a write action, using the `ApplicationManager`'s `runWriteAction()` method.

Figure 4.1 illustrates how this synchronization works internally. More specifically, if a background thread attempts to explicitly access IntelliJ's data structures, then an exception will be thrown. The proper way to access those structures is to issue a read request. A corresponding entry is added in an event queue, which is processed through a central event loop. The thread is blocked until its request is handled to completion. In contrast, the UI thread can read directly, without going through the event loop.

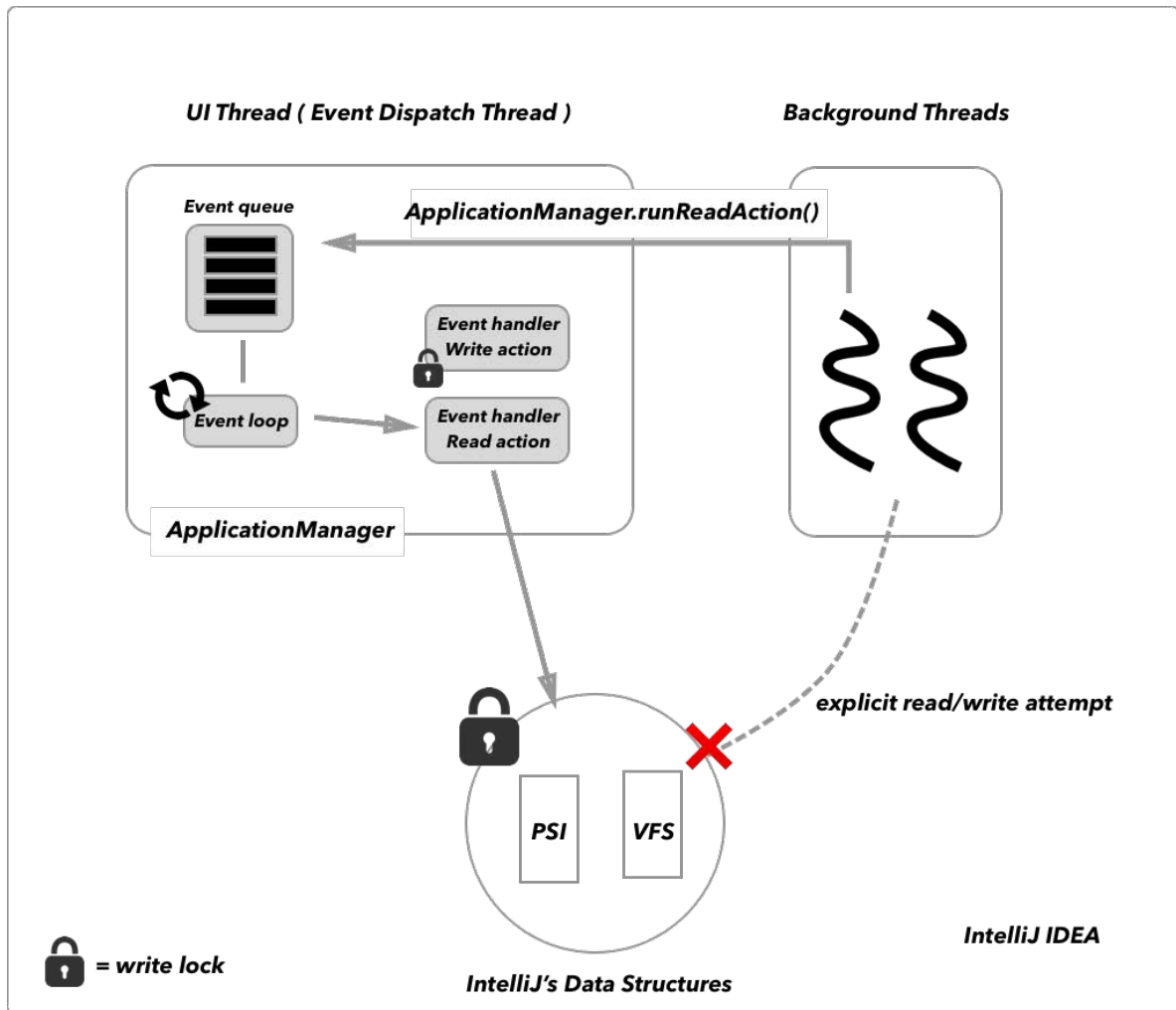


FIGURE 4.1 - THREAD MODEL

4.3 PSI

The Program Structure Index, commonly referred to as PSI, is the layer in the IntelliJ Platform that is responsible for parsing files and creating the syntactic and semantic code model that enables many of the platform's features.

A PSI file represents a hierarchy of PSI elements (so-called PSI trees). A single PSI file may include several PSI trees in a particular programming language. A PSI element, in its turn, can have child PSI elements.

PSI elements and operations on the level of individual PSI elements are used to explore the internal structure of source code as it is interpreted by the IntelliJ platform. Thanks to the hierarchical structure, it is easy to navigate between PSI elements and perform modification on elements instead of trying to modify the contents of a file at a low-level using byte-streams.

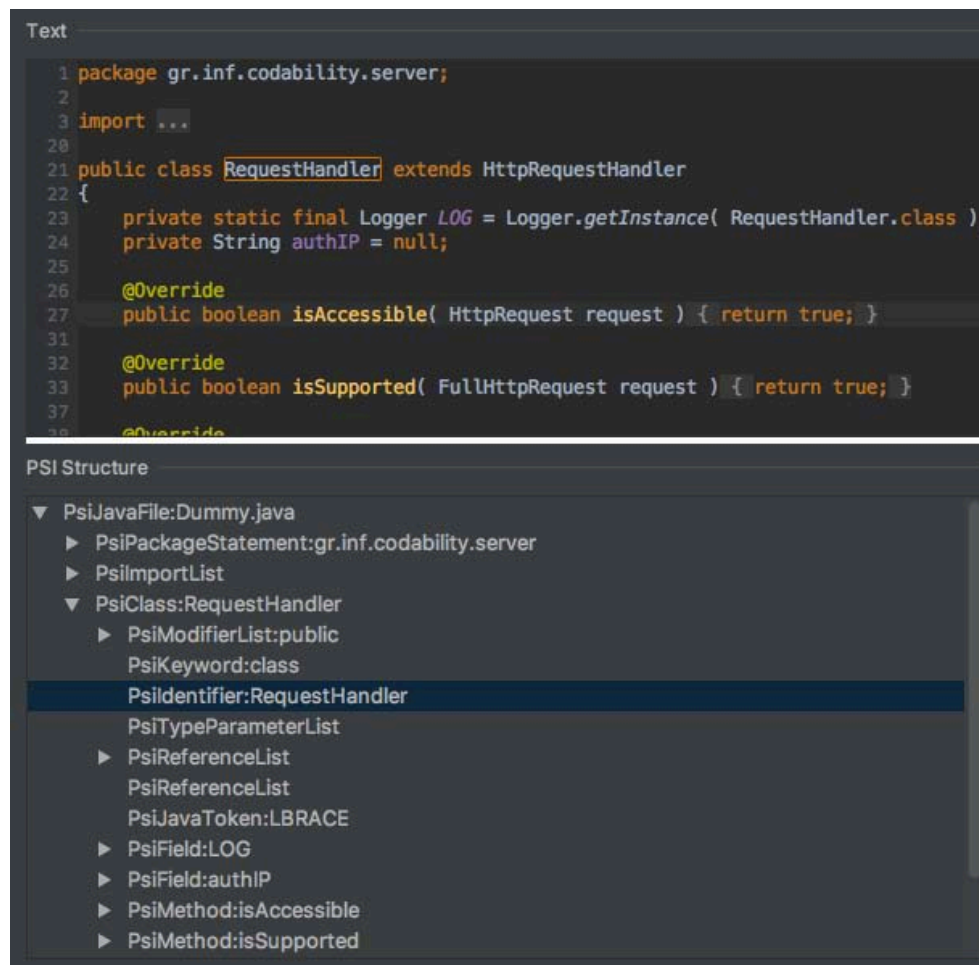


FIGURE 4.2 - EXAMPLE OF A PSI TREE

Figure 4.2 shows a PSI tree created after parsing the contents of a Java file. Every Java statement is actually transformed into a PSI element. For example, as highlighted in the figure above, class `RequestHandler` is recognised as a `PsiClass`, with several attributes as its children.

When it comes to the modification of the PSI tree, the procedure is rather trivial. For instance, a method can be added inside a specific class merely by traversing the PSI tree from its root, until the relevant PsiClass is found. Then the method can be appended as a PsiMethod child of the PsiClass node.

5.1 Overview

This chapter introduces the core functionality of the Codability plug-in. Listing 5.1 provides a high level overview of how the plug-in works. Next, the most important elements of the implementation are discussed in more detail.

Simplified flow of Codability's functionality

1. receive data
 2. parse data
 3. extract/match command type and parameters
 4. call function that modifies the PSI tree
-

LISTING 5.1 - CODABILITY'S FUNCTIONALITY

5.2 Web server

The first step is to receive the data sent by the mobile device. The initial plan was to implement a custom server that would listen to a TCP/IP socket and handle the requests carrying user commands in textual form, sent by the mobile application. This approach was quickly abandoned, as the IntelliJ platform has a built-in web server, mostly for debugging purposes.

In order to take advantage of this feature, an extension for this web server has been developed. The corresponding request handler listens for HTTP requests received by the web server and then redirects them to the plug-in for further processing.

5.3 Command hierarchy & syntax

Codability organizes user commands in a hierarchy, displayed in Figure 5.1. This approach facilitates the parsing of commands and their parameters. It also makes it easy to extend Codability by introducing new commands that provide additional functionality.

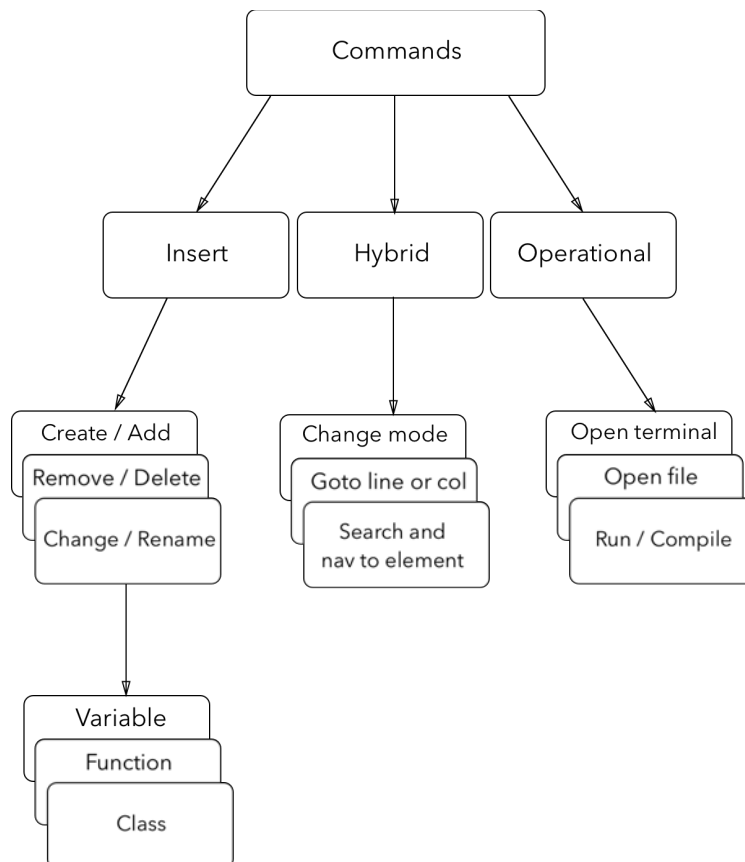


FIGURE 5.1 - COMMANDS HIERARCHY

At the top level, commands are classified into three different categories, depending on their context/mode:

- Insert: As the name suggests, such commands serve the purpose of writing code by adding or modifying elements on the PSI tree. Commands that belong to these category, are "Create Class", "Create Function", etc.

- Operational: These commands are related to the control the IDE. Such commands are "Create Project", "Open file", "Execute", etc.
- Hybrid: This mode refers to commands that can run on both of the above mentioned contexts. Commands in this category include "Go to line", "Search for", "Mode change", etc.

This classification has been made to separate commands that have totally different context. In addition to this, by adopting this approach, less keywords are reserved for each mode, thus enabling more flexibility in case of additional extensions that need to be introduced in the future.

In terms of syntax, the first word of a command consists of a reserved keyword, which identifies a particular function from one of the three top-level categories. The words that follow are usually key-value pairs, corresponding to the parameters of that particular function.

The reserved keywords are divided into two tiers. Tier 1 is for the first dictated word, whereas Tier 2 refers to words used as keys for commands that need extra parameters. Table 5.2 lists all reserved keywords.

Tier 1	Tier 2
Create / Add	Class
Delete / Remove	Constructor
Open	File
Print	Function
Run	Name
	Type
	Variable

TABLE 5.1 RESERVED KEYWORDS

5.4 Command decoding

After performing a simple syntax analysis and tokenising the command into separate words, the syntax rules are applied initially to check whether the command is in a valid format. If it this check is successful, the type of the command is determined by comparing the first two words of the command with some pre-defined templates to decide whether they match a defined command type. If so, the parameters of the command are extracted and checked for validity. Finally, based on the command type, the command's parameters and the format of the command, the so-called execution unit is invoked in order to call the proper function that will modify the PSI tree accordingly.

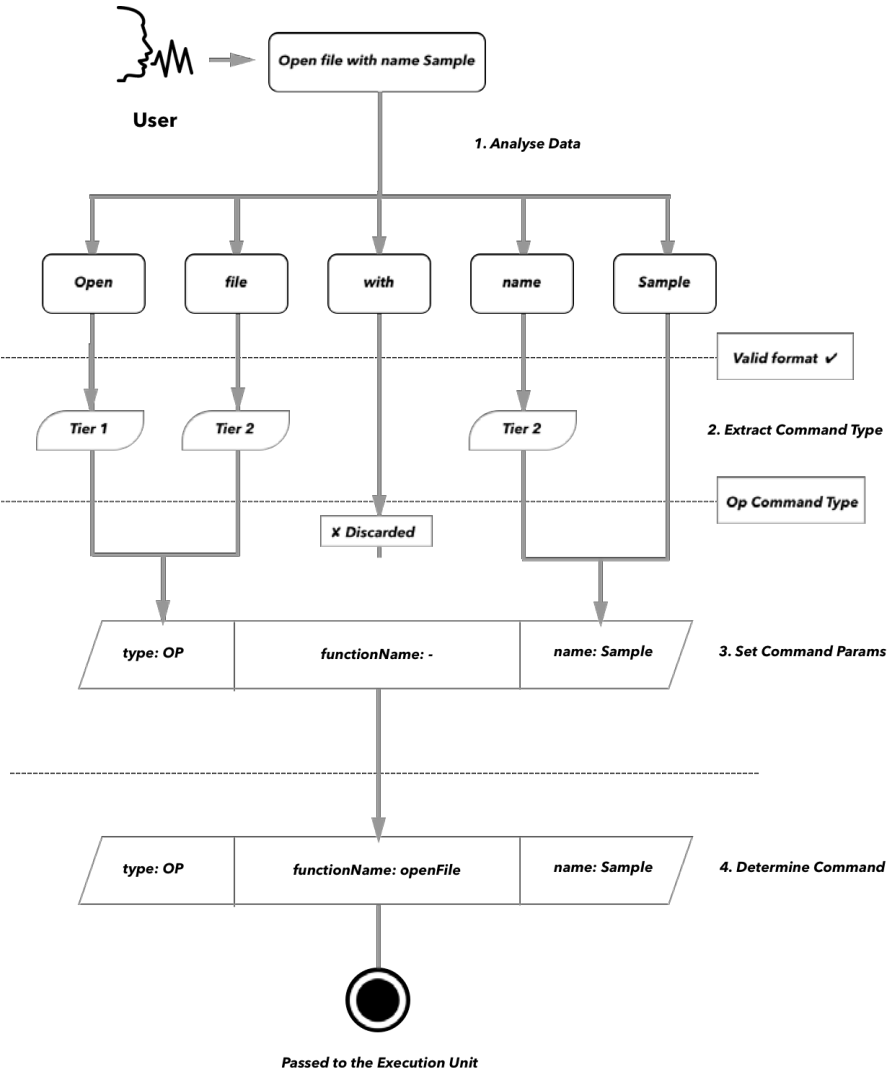


FIGURE 5.2 - COMMAND DECODING EXAMPLE

Figure 5.2 shows this process for the case where the user dictates the command "Open file with name Sample".

At the beginning the command is being tokenised into individual words. This allows the decoder to perform a validity check by comparing the words to test if they match one of the pre-defined templates. In this example the correspondent template would be "Open file [...] name fileName". Therefore, the command is being approved and then the command type is being decided.

In this stage the first two tokens of the command define the command type. In this case "Open file" belongs to the "OP" mode, which refers to actions that are relative to the control of the IDE. Following this, any other words that match the optional "[.]" category are being discarded and any other words that match the command format, form key-value pairs to be used as the commands parameters. For instance, the "with" token is being discarded and the "name" and "Sample" tokens form a key-value pair, "name: Sample".

Having extracted the command type and the command parameters, an instance of the Command class is created and those variables are assigned to the command.

The final stage is to determine the function name of the command in order to be recognised by the execution unit. This is accomplished by checking the contents of the Command variable. A command with identifier "Open file", command type "Op" and a parameter key "name" maps to the function name "openFile". Once the function name is being assigned, the command variable is ready to be passed to the execution unit for execution.

5.5 Command execution

After the decoding process has been completed, the command is passed to the execution unit. The execution unit matches the command's function name with the actual implementation of that function and starts executing the command

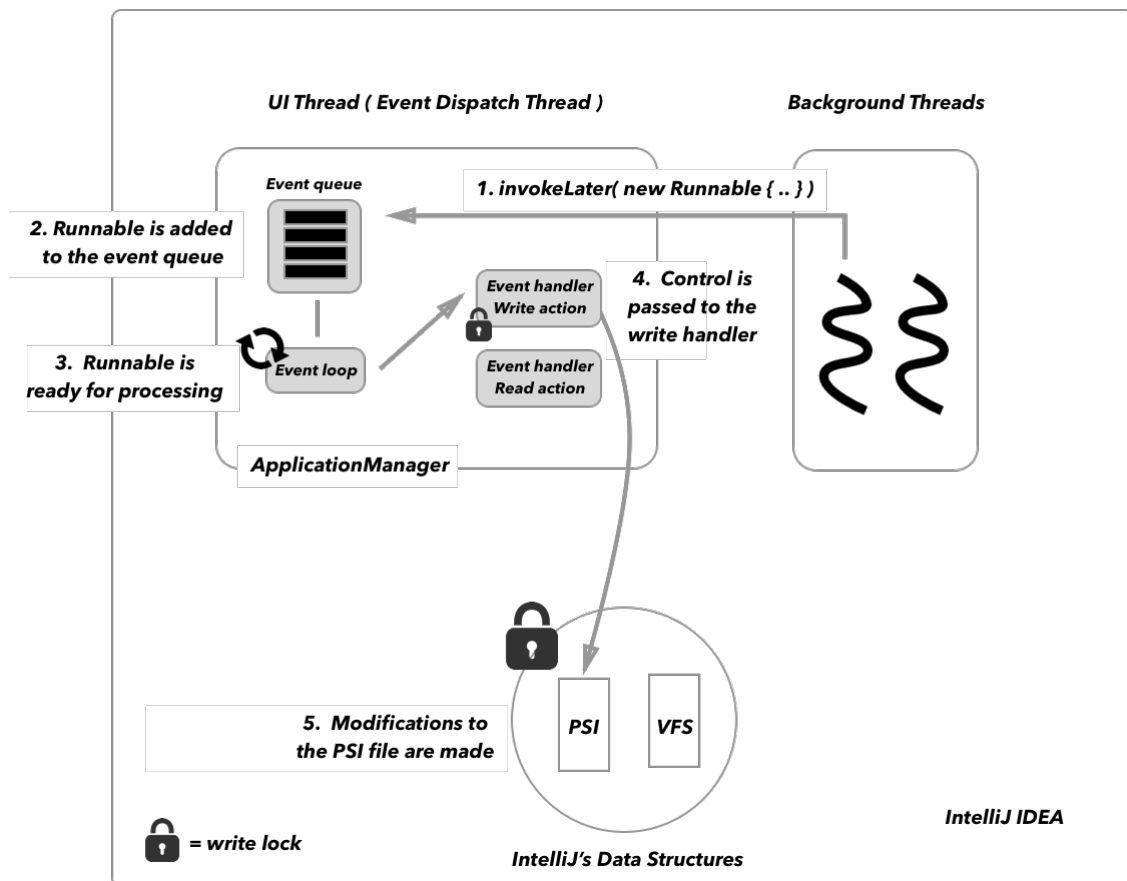


FIGURE 5.3 - EXECUTION PROCESS

The execution process most of the times involves modifying the PSI tree. For this to happen, given that the Codability plug-in runs as a background thread (not within the UI thread), every command function needs to be wrapped in a `Runnable` instance and passed as an argument to `ApplicationManager`'s `invokeLater()` method, which causes the `Runnable` to be executed asynchronously on the event dispatching thread. The execution will happen after all pending events have been processed.

```
ApplicationManager.getApplication().invokeLater() { new Runnable ... }
```

When the function is eventually invoked, as a first step, it gets an instance of the currently opened project and also an instance of the IDEA text editor which holds informations about the opened files.

```
Project project = ProjectManager.getOpenProjects()[ 0 ];  
Editor editor = FileEditorManager.getSelectedTextEditor();
```

Afterwards, the PSI file needs to get retrieved, so that modifications on the PSI tree can be made:

```
JComponent component = editor.getComponent();  
DataContext data = DataManager.getDataContext( component );  
PsiFile psiFile = DataKeys.PSI_FILE.getData( dataContext );
```

From this point onwards, the actions to be performed depend on the PSI elements that need to be added or modified, according to the type of command and function invoked. Generally, PSI elements are created by via the `PsiElementFactory` class, which features several methods to add variable elements to the PSI file.

```
PsiElementFactory factory = JavaPsiFacade.getElementFactory( project );  
PsiMethod method = elementFactory.createMethodFromText( ... );
```

Finally, each action that modifies the contents of the PSI tree needs to be run under write-safe context, by wrapping it within a `WriteCommandAction` action.

```
new WriteCommandAction.Simple( project ) {  
    @Override  
    protected void run() throws Throwable { ... }  
}.execute();
```

5.6 How to extend the plug-in

The Codability plug-in is structured so that a developer can extend its functionality with little extra work. In order to add an extra command, the developer must follow the guidelines mentioned in Figure 5.4.

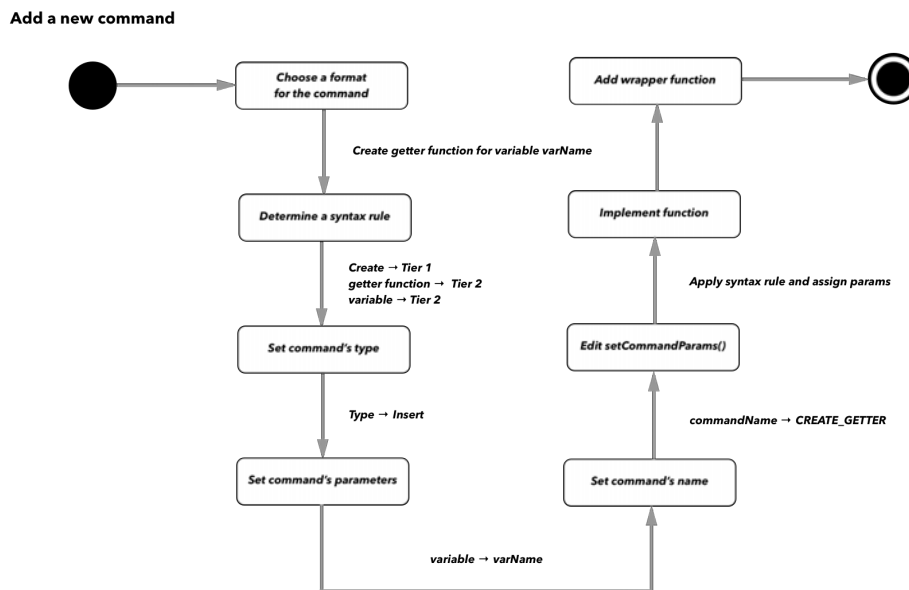


FIGURE 5.4 - PLUG-IN EXTENDING PROCEDURE

Step 1: The developer has to define the format of the command. He has to decide the purpose of the command, its content and structure and in which context it should be run. Suppose, for example's sake, that a developer wants to create a command that automatically creates a getter function for a variable. He chooses the following format for this command "Create getter function for variable varName". As this command indicates an addition to the PSI tree, the proper context in which this command should run is the insert mode.

Step 2: A syntax rule should be determined for the new command. The developer has to choose which words will be needed as Tier 1 and Tier 2 keywords. Obviously, "create" should be set as a Tier 1 keyword, "variable" and "getterFunction" as Tier 2 keywords.

Step 3: The developer has to add the name that corresponds to the function inside the `CommandName` class of the `gr.inf.codability.command` package. In this example, the name to be added is "CREATE_GETTER".

Step 4: The `Command` class of the `gr.inf.codability.command` package has to be modified as well. More specifically, the developer should edit an `else-if` statement in the `setCommandParams()` method to comply with the syntax rules and assign a `CommandType`, a `CommandName` and a `setParams()` method with the proper Tier 2 keyword as an argument.

```
else if ( command( "create getter function" ) ) {
    name = CREATE_GETTER;
    type = INSERT;
    setParams( "variable" );
}
```

Step 5: The developer should go to `gr.inf.codability.function` package, create either a new class or edit an existing one in order to add the implementation for this function.

Step 6: A corresponding wrapper function has to be added in the `CommandImpl` class of `gr.inf.codability.command` package:

```
static void createGetter( HashMap<String, String> nameOfVar ) {
    if ( nameOfVar.get( "variable" ) == null )
        return;

    String name = getVariableNameFormat( nameOfVar, "variable" );
    createGetterFunction( name ); // refers to the function package
}
```

Step 7: Inside the same package but in the `Command` class, under the `execute()` method the following snippet must be added:

```
else if ( name == CREATE_GETTER )
    createGetter( params );
```

To get a feeling about how useful the system actually is in practice, a few simple experiments were performed to test its accuracy and the end-user's performance. The measurements were conducted in a silent environment in order not to obstruct the speech recognition process. The lack of strong ambient noise is a rather reasonable assumption for most software developers, especially ones that work at home.

Accuracy

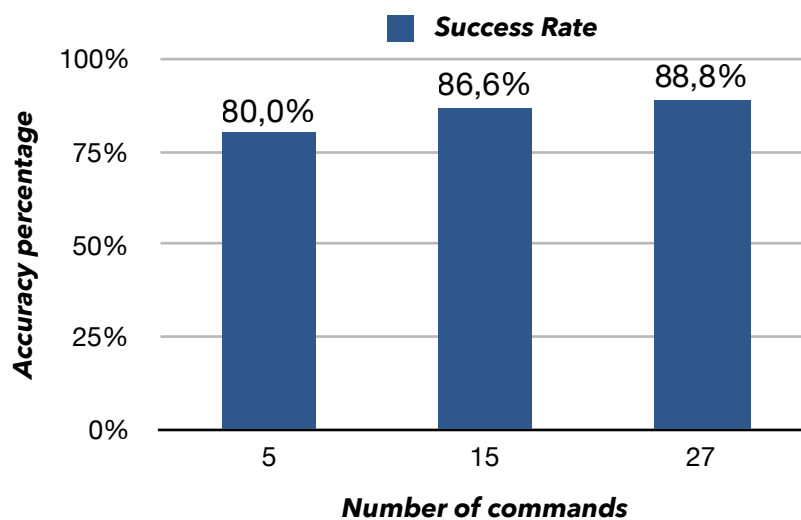


DIAGRAM 6.1 - COMMAND ACCURACY

Diagram 6.1 shows the number of commands that were accurately converted by the speech recognition service, as a function of the number of commands that were issued. The accuracy of Google's Speech API after 15 commands capped at a maximum of 88%. The diagram also shows that as the number of commands increases, the accuracy of the service improves. This is presumably due to the application of Google's internal deep learning algorithms, which adapt the speech recognition to the context of the previous converted commands within a speech recognition session.

Although 88% may not be very satisfactory for open-ended speech, it proved to work quite well for the purposes of our oral programming interface where the vocabulary is much more restricted. It is expected that as technology advances, the accuracy percentage will reach even higher numbers.

Performance

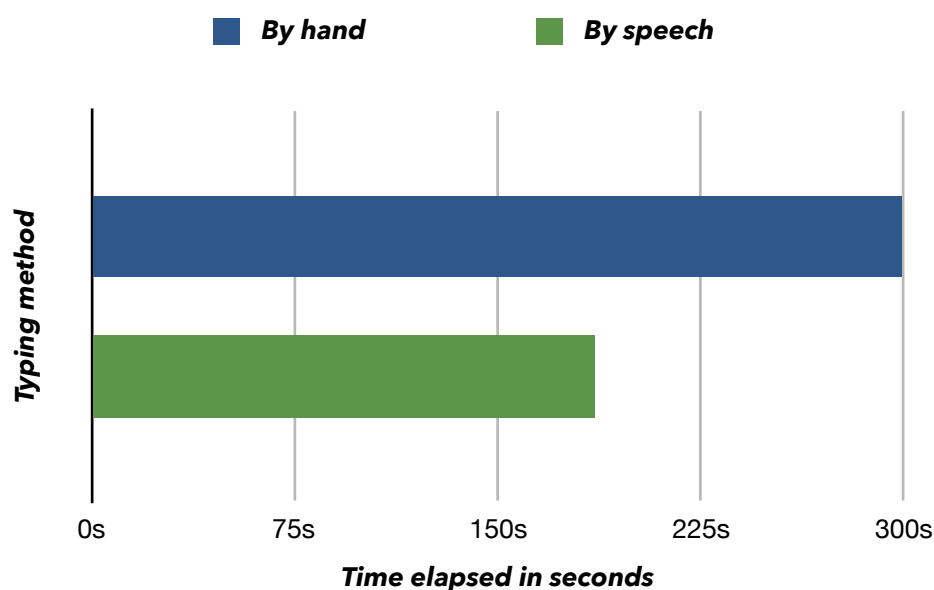


DIAGRAM 6.2 - CODING PERFORMANCE

A measurement of the time required to write a simple Java application consisting of 799 characters (~700 keystrokes using auto-completion) had been taken. As a reference, the same application was written by hand using auto-completion by the author, who has a physical disability, resulting in slower than usual typing speeds. Coding by speech via the Codability plugin was faster, roughly by a factor of 1.6x compared to coding using a keyboard. This is a significant improvement. Moreover, the author found the oral interface less stressful compared to a more conventional input method involving hand movement.

CONCLUSION

Our work provides a proof of concept, showing that coding by speech using an oral programming interface is feasible and can result in a speedup of typing performance, especially for programmers with a form of a physical disability. We believe that oral programming has the potential to become, if not the main coding method, a very attractive alternative method for programmers.

The Java language is a language with literally thousands of methods, classes and libraries. While it was clearly impossible to cover all of them, our work demonstrates that this can be done incrementally and in a structured way. At this stage, Codability supports only some basic features of the Java language, and an obvious next step would be to add and implement more aspects of the Java Language, or target other popular languages such as C and Python.

From an algorithmic point of view, the current syntax could be modified to support a more free form speech. Allowing the developer to dictate tasks in a more natural way would most certainly increase productivity. Ideally machine learning methods could be applied to better predict the user's intentions.

Finally, a more generic version of the mobile application could be developed in order to be compatible with a wider range of mobile devices and smartphone operating systems.

BIBLIOGRAPHY

[1] <https://www.jetbrains.org/intellij/sdk/docs/welcome.html> , **JetBrains IntelliJ Platform SDK.**

[2] <https://upsources.jetbrains.com/> , **IntelliJ's source code**

[3] <https://intellij-support.jetbrains.com/hc/en-us/community/topics/200366979-IntelliJ-IDEA-Open-API-and-Plugin-Development> , **IntelliJ IDEA Open API and Plugin Development Forum**

[4] Krochmalski, J. (2014). IntelliJ IDEA essentials. 1st ed. , **IntelliJ IDEA tips & tricks**

[5] Marsicano, B. (2017). Android Programming. 3rd ed.: Big Nerd Ranch Guides , **Android Programming**

[6] <https://developer.android.com/index.html> , **Android SDK**

[7] Yener, M. and Dundar, O. (2016). Expert Android Studio. 1st ed. , **Plugin tutorial**

A.1 Setting up the development environment

This section is addressed to developers who are willing to develop and extend more the functionality of the plug-in. It features a mini-guide to setup the development environment.

The first step is to download and install the latest Java SDK from <http://www.oracle.com/technetwork/java/javase/downloads/>

Java SE Development Kit 8 Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, applets, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.

See also:

- [Java Developer Newsletter](#): From your Oracle account, select **Subscriptions**, expand **Technology**, and subscribe to **Java**.
- [Java Developer Day hands-on workshops \(free\) and other events](#)
- [Java Magazine](#)

[JDK 8u144 checksum](#)


Java SE Development Kit 8u144

You must accept the [Oracle Binary Code License Agreement for Java SE](#) to download this software.
Thank you for accepting the [Oracle Binary Code License Agreement for Java SE](#); you may now download this software.

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.89 MB	jdk-8u144-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.83 MB	jdk-8u144-linux-arm64-vfp-hflt.tar.gz
Linux x86	164.65 MB	jdk-8u144-linux-i586.rpm
Linux x86	179.44 MB	jdk-8u144-linux-i586.tar.gz
Linux x64	162.1 MB	jdk-8u144-linux-x64.rpm
Linux x64	176.92 MB	jdk-8u144-linux-x64.tar.gz
Mac OS X	226.6 MB	jdk-8u144-macosx-x64.dmg
Solaris SPARC 64-bit	139.87 MB	jdk-8u144-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.18 MB	jdk-8u144-solaris-sparcv9.tar.gz
Solaris x64	140.51 MB	jdk-8u144-solaris-x64.tar.Z
Solaris x64	96.99 MB	jdk-8u144-solaris-x64.tar.gz
Windows x86	190.94 MB	jdk-8u144-windows-i586.exe

FIGURE A.1 - DOWNLOAD JAVA SDK

The next step is to visit <https://www.jetbrains.com/idea/download/> and download and install the Community version of IntelliJ IDEA.



Download IntelliJ IDEA

Windows macOS Linux

Ultimate

For web and enterprise development

DOWNLOAD

Free trial

Community

For JVM and Android development

DOWNLOAD

Free, open-source

Version: 2017.2.2
Build: 172.3757.52
Released: August 15, 2017

[System requirements](#)
[Installation Instructions](#)
[Previous versions](#)

FIGURE A.2 - DOWNLOAD INTELLIJ

After the installation of IntelliJ IDEA has been completed, following procedure needs to be applied. From the startup screen, “Checkout from Version Control” needs to be selected and the link below should be added [git://git.jetbrains.org/idea/community.git](https://git.jetbrains.org/idea/community.git)



FIGURE A.3 - CHECKING OUT INTELLIJ'S SOURCE CODE FROM GIT

When the cloning of the repository has finished, the project should be opened and the command `./getPlugins.sh` should be entered.

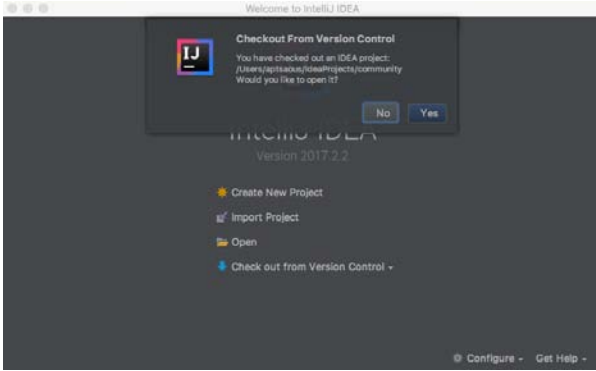


FIGURE A.4 - CHECK-OUT COMPLETE

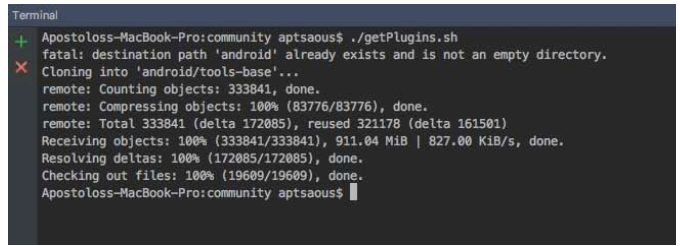


FIGURE A.5 - RUN ./GETPLUGINS.SH

Finally, a JSDK named "IDEA jdk" (case sensitive) needs to be configured, pointing to an installation of JDK 1.8. This can be done by going to the menu under File → Project Structure

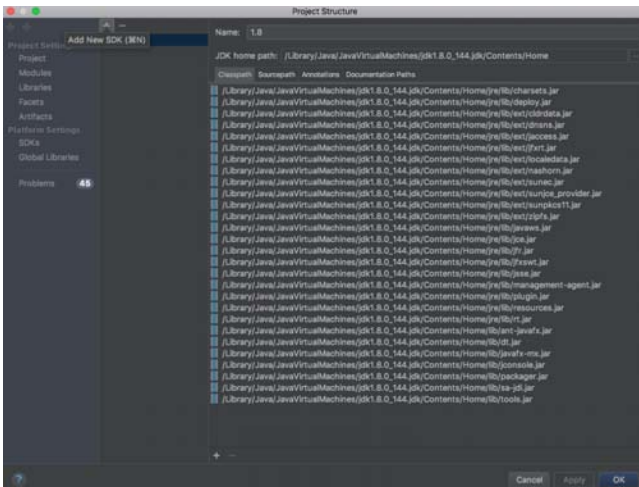


FIGURE A.6 - ADD NEW SDK

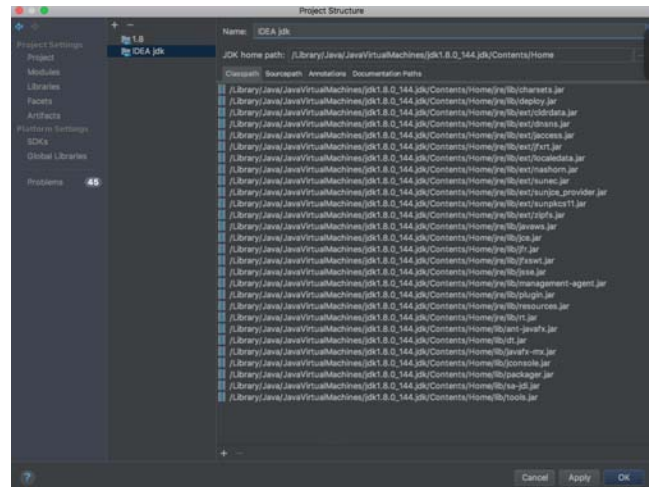


FIGURE A.7 - SAVE NEW SDK

A.2 Codability's source code

The source code of the Android App can be found at <https://github.com/aptsaous/CodabilityApp>, whereas the source code of the plugin can be found at <https://github.com/aptsaous/Codability>