HYPERCODE – VOICE AIDED PROGRAMMING

Rinor Maloku

Follow this and additional works at: https://knowledgecenter.ubt-uni.net/etd

Part of the Computer Sciences Commons
Department of Computer Science and Engineering

HYPERCODE – VOICE AIDED PROGRAMMING

Bachelor Thesis

Rinor Maloku

October/2016

Prishtinë
This thesis work was prepared and submitted in partial fulfillment of the requirements for the Bachelor degree.
ABSTRACT

Existing input methods for inserting text into digital devices are slow and have a steep contrast of information exchange difference between human and computer. While programmers quickly retrieve information from the computer visually, via the screen, it is a slow process of inserting information via mouse and keyboard. Speech Recognition can be incorporated as another input method to enable inserting continuous text into the computer for increased speed.

Accuracy improvements of speech recognition is driven by incorporation of deep neural networks and thus made speech recognition another viable input method, which is used in smartphones, computers and smart homes, to enable hand free technology usage by providing a voice user interface.

This thesis presents HyperCode, a framework that makes use of Speech Recognition as another input method to increase coding efficiency, in the Java programming language, by using already present libraries that enable modification, extraction and alteration of the recognized words.
ACKNOWLEDGEMENT

The path during these years was difficult and full of obstacles, but each obstacle passed was a lesson learned, a skill acquired, a professional and personal development. It wouldn’t have been possible without the people that were involved and made those years and those challenges endurable. I want to express my gratitude and profound respect to my parents Sahadi and Sheride Maloku and my brothers Aurel and Drilon, with your support everything is possible.

Best regards to my Professor and supervisor Dr. Bertan Karahoda for advices, encouragement and support.

I am thankful to University for Business and Technology (UBT) for the opportunities, comfortable conditions provided, the expertise and the academical support available when needed.

Lastly I want to express the deepest gratitude for the people that distinguished my abilities and gave me the exposure that lead to a better future for me, those people are: Tringa Hana, Muhamed Retkoceri, MSc. Besart Plana and MSc. Shkelqim Berisha.
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
</tr>
</tbody>
</table>

## INTRODUCTION

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

## LITERATURE REVIEW

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Speech Recognition</td>
<td>2</td>
</tr>
<tr>
<td>2.2 Programming Concepts</td>
<td>3</td>
</tr>
</tbody>
</table>

## PROBLEM STATEMENT

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

## METHODOLOGY

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

## HYPERCODE – VOICE AIDED PROGRAMMING

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Dragon NaturallySpeaking</td>
<td>7</td>
</tr>
<tr>
<td>5.2 Natlink</td>
<td>8</td>
</tr>
<tr>
<td>5.3 Dragonfly</td>
<td>8</td>
</tr>
<tr>
<td>5.4 IntelliJ IDEA 2016</td>
<td>9</td>
</tr>
</tbody>
</table>

## STATE OF THE ART

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Clarifications on time measurement</td>
<td>11</td>
</tr>
<tr>
<td>6.2 The tasks</td>
<td>12</td>
</tr>
<tr>
<td>6.2.1 Creating the Book class</td>
<td>12</td>
</tr>
<tr>
<td>6.2.2 Creating the new Book class</td>
<td>15</td>
</tr>
<tr>
<td>6.2.3 Creating primitive attributes</td>
<td>16</td>
</tr>
<tr>
<td>6.2.4 Creating object attributes</td>
<td>17</td>
</tr>
<tr>
<td>6.2.5 Creating the class constructor</td>
<td>18</td>
</tr>
<tr>
<td>6.2.6 Creating getter and setter methods</td>
<td>20</td>
</tr>
<tr>
<td>6.2.6.3 Time measurements</td>
<td>21</td>
</tr>
<tr>
<td>6.2.7 Generating the Equals method</td>
<td>21</td>
</tr>
<tr>
<td>6.2.8 Creating the toString method</td>
<td>23</td>
</tr>
</tbody>
</table>
CONCLUSION .......................................................................................................................... 25
APPENDIX ................................................................................................................................ 26
  Additional Features .................................................................................................................. 26
REFERENCES ............................................................................................................................ 27
Bibliography .............................................................................................................................. 27
LIST OF FIGURES

Fig. 1. The process of turning a voice command into code................................. 7
Fig. 2. Result for dictation “public class book”.................................................. 14
Fig. 3. Result for dictation “public class book extends abstract book implements readable 
and serializable” .............................................................................................. 14
Fig. 4. Result for dictation “new class book” in this case the file was created too ....... 16
Fig. 5. Result for dictation "new enum book genres" ........................................ 16
Fig. 6. Result for dictation “Private int number of pages” ................................. 17
Fig. 7. Result for dictation "public boolean is out of print" ............................... 17
Fig. 8. Result for dictation “private string instance author” .............................. 18
Fig. 9. Result for dictation "private big decimal instance price of cost" ............... 18
Fig. 10. Result for dictation “Add constructor” while inside a class Book with two 
attributes authorname and number of pages.................................................... 18
Fig. 11. Result for dictation “Add constructor” while inside a class Player with two 
attributes name and score ................................................................................. 19
Fig. 12. Code of class Book................................................................................. 19
Fig. 13. Result for dictation “Get author name”.................................................. 20
Fig. 14. Result for dictation “Generate equals method” ....................................... 21
Fig. 15. IntelliJ’s UI for selecting options available for an object ......................... 22
Fig. 16. Result of the sequence of commands in section 6.2.7.3......................... 22
Fig. 17. Result of the sequence of commands in section 6.2.8.1......................... 23
1 INTRODUCTION

We are striving productivity since the advent of technology, we are on the lookout for the best software for efficient task completion, we are using single purpose devices optimized for specific jobs, computers for diverse needs and notebooks when on the go, wearable devices to get important notifications, clouds of big data and the internet of things. In the last decades the flux of information exchange between machines intensified, meanwhile it’s exchange between human and machine is still the same: keyboard and mouse, and for devices without those we have touchscreen that simulate keyboards and graphical user interface control. Convenient but comes with 80% reduction of possible efficiency.

The perfect solution without drop of efficiency comes in the form of a technology that was being developed since the beginning of the second half of the 20th century and for humans it is the most natural way of communication, speech. Speech Recognition is another input method that enables information exchange between Human-Machine. It is finding usage in devices where keyboard usage is impractical or inefficient it furtherly helps automation of computer tasks, enabling us to sleep our devices, wake them up, inquire about the weather and dictate the news.

Those automation features can benefit programmers with reduction of time needed to write code, and as proven it can be an alternative for programmers to code by voice when disabled by upper limb disorders.

In this thesis we will cover relevant information for Speech Recognition, basics of programming that will be shown as examples later, state of the art of the framework and the time that can be reduced when using the features at its optimum.
2 LITERATURE REVIEW

2.1 Speech Recognition

Speech recognition is an input method that enables computers to receive spoken information and to respond accordingly. As speech is the most natural way of communication between humans a similar interaction with machines was desirable and a prominent theme in the science fiction culture. Computer Scientists saw SR as the next step in this human-machine interaction and constantly pushed the boundaries by developing algorithms and systems for its recognition.

The first Speech Recognition system was developed in the Bell Labs in 1953, and could recognize numbers "zero" to "nine" for only one speaker. Speech recognition advanced immensely in the following decades and became one of the most prevalent technologies in the 21st century, with purposes as various as browsing the web, making an appointment, typing emails, controlling smart-homes etc. [1]

Early SR software were limited in the number of words they recognized, had high word-error rate, lacked continuous speech recognition capabilities and as such its application was more a gimmickry than real usage. In the late 1970s SR software made a move from template and simple pattern recognition approach to the statistical approach (Hidden Markov Model) introduced by L. E. Baum, which resulted in a major word recognition improvement and drastic drop of word-error rate. [2] Furtherly improvements of algorithms and usage of existing ones sped up recognition, Bamberg al present the Rapid Matcher component that alongside HMM use Dynamic programming “to prune all paths that score poorly relative to the best path” which sped up recognition by 10 times, and that’s “how real time recognition was achieved” [3].

Another push of further improvement of Speech Recognition technologies came with the increasing usage of Virtual Speech Assistants which being ubiquitous provided big data and with big data the ground for deep neural network usage reducing error rate by 1/3 over previous methodologies. [4]
Another important technology in SR is continuous speech recognition which enables the user to chain a sequence of words without in-between pauses, reducing time of triggering commands and making human-machine communication by speech more natural.

2.2 Programming Concepts

To understand the functionalities provided by our framework understanding of below concepts is a requisite:

- Computer program is sequence of statements that lead to a desired outcome (Solution of a problem).
- Statement is an instruction for the computer to perform an action.
- Object oriented programming: is a programming paradigm based on objects, made up from attributes and methods defined by Classes of objects.
- Boilerplate code repetitious code that has to be include with little alteration.
- Attributes usually are comprised of the access modified, its type and name.
- Access modifiers determine the accessibility of an attribute from another class.
- The type of an attribute can be one of the basic primitive types or it can be a combination of those.
- A method in Java is made up of, its access modifier (has the same function as for the attribute), return type, method name, parameters and its body.
3 PROBLEM STATEMENT

Static typed programming languages like Java contain a lot of ceremonial code. This makes typing on the keyboard a much more intensive task. For programmers in a lengthier period of time, this could introduce Upper Limb Disorders (ULD), which can disable them from keyboard usage and thus from programming, disadvantageous for both employer and employee. According to Hanson MA et al (1999) at an average 14% of keyboard operators report to have had severe ULD symptoms recently [5].

In another study, Palmer et al (2001) came to a similar result where 13.3% of keyboard users have had ULD symptoms and concluded that one-third of them had symptoms as severe as to prevent their normal activities [6]. Considering that presence of ULDs causes the lengthiest work absences, like for example Carpal Tunnel Syndrome being the most extreme with a median of 32 workdays [7], a solution for the individual and the corporation is essential.

As an alternative input method Voice Speech Recognition could be used to reduce keyboard reliance up to the point of replacing it. Patel and Patel proposed a system that generates code from voice input into their developed editor. Afterwards the user pastes the generated code into the desired Java Integrated Development Environment [8].

Masuoka gives another example of programming in Java by voice input by developing an extension for VoiceCode. VoiceCode is a software designed to be used in combination with Dragon NaturallySpeaking, a speech recognition program, and Emacs, an editor [9].

Our framework HyperCode is comprised of Python scripts to enable coding in Java using voice commands that interface directly with the Java IDE. HyperCode is specifically built to be used with IntelliJ IDEA with the purpose of enabling programmers to use the IDE features and simplifying navigation in a complex development environment.
It has to achieve three goals:

- **Map Java keywords and common code snippets with voice commands:** Achieved when the framework enables programming only by voice input, which would make programming possible for people suffering from ULDs or other similar disabilities.

- **Controlling IntelliJ IDEA interface using voice commands:** IntelliJ IDEA has an extensive hot-key list that enables a complete control of functionalities.

- **Reducing the overall time needed for writing code:** Incorporate Speech Recognition as a third input method with the purpose of speeding up information exchange between programmer and computer, by replacing the typing of ceremonial code of the Java programming language with voice commands.

Achievement of these three goals would lead to a framework that is easier to work with, faster alternative when used at its optimum and features provided by other voice coding projects would be a subset of it’s features.
4 METHODOLOGY

The methodology for this thesis is Research and Implementation. As a basis for the research we took written books in the subject of Speech Recognition. While for implementation we build upon libraries that already enabled usage of voice commands to perform actions on the computer. In order to conclude about the success of the developed framework for coding by voice an experiment was performed with the most basic structure in Java that of a simple object and the class for it. Times were compared in order to come to the outcome of benefits in efficiency.

The measurements from the benefits were done by comparing optimal usage of different techniques described in the sections to come.

Dragon Naturally Speaking was used as the Speech Recognition software in order to reduce recognition error rate as it can adapt to the dialect of the speaker. Training of DNS is done by a static text it provides which the user has to read and it improves accuracy of recognition.

The framework was developed using the Python programming language, and built to enable programming in the Java programming language, but posses the possibility of using already defined methods to bootstrap creation of Grammar Objects to include other programming languages.
5 HYPERCODE – VOICE AIDED PROGRAMMING

To turn a voice command into code we need Speech Recognition provided by Dragon NaturallySpeaking and mapping of commands to custom content provided by Natlink. Additionally open source libraries from the python programming community were used with special importance being Dragonfly, which offers rich but simple to use tools for complex functionalities which are used by HyperCode for content manipulation, hot-key simulation and compound actions.

![Diagram of voice command to code process]

Fig. 1. The process of turning a voice command into code

5.1 Dragon NaturallySpeaking

Dragon NaturallySpeaking (DNS) is a speech recognition software which offers an extensible vocabulary and enables differentiation from spoken and written form. This is convenient as words can be preserved and situations of confusion evaded, for instance when the user wants to write “print to console” or wants to call a command to print to console. Furtherly DNS has continuous speech recognition capabilities which are used throughout our scripts to enable complex grammar objects to be created and used for different purposes by reacting differently to the content of the grammar object.

DNS is based on Microsoft Speech API which in turn is based on Microsoft Component Object Models. For this work the most important components are:

Voice Dictation API – high-level objects for continuous dictation speech recognition
Direct Speech Recognition API – used to add words and create grammars. When a specific recognition happens clients are informed.

Speech Service API is another COM interface that enables sending keystrokes to a third party application [10].

5.2 Natlink

Natlink is a DNS extension which exposes its APIs to Python scripts, it enables Grammar Object creation needed for speech recognition to be performed. After speech recognition when one of the Grammars is recognized the appropriate Python function gets called [10].

Natlink extension exports functions of Speech Service API, most notable Natlink’s playString function that takes a String and writes its content to the currently focused application [11]

The most important aspect to understand about Natlink is that of the Grammar Objects which is more or less a structure similar to the “Hashtable” datastructure, but in this case both the key and the value are dynamic, meaning that the contents of the dictation can vary and the same grammar object will be activated which in turn executes scripts that produce an output that varies according to the dictation. Grammar Objects will be explained with examples in next chapters.

5.3 Dragonfly

Dragonfly is a Python library that is designed to allow speech commands and grammar objects to be treated as first-class Python objects [12].

It’s main features are:

- **Language Object Model**: Simplifies defining and retrieving information from recognized dictation. It’s foundation is “....a flexible object model for handling speech elements and command grammars” [12].
• **Support for multiple speech recognition engines**: Scripts developed with Dragonfly will work in Dragon NaturallySpeaking and Windows Speech Recognition.

• **Built-in action framework**: Powerful framework for executing actions. Described furtherly below.

**Classes from Dragonfly used throughout our framework:**

• **Dictation**: takes a String as an argument which represents a sequence of speech. That can be passed later as an argument.

• **Choice**: takes a Map as a parameter that can retrieve value when specific key in the map is recognized. Improves recognition for words that would initially be of lower priority.

• **Pause**: pauses execution of macros for milliseconds provided as a parameter.

• **WaitWindow**: pauses execution and waits for the window with the specified title provided as a parameter.

• **Function**: takes a function reference, enabling extraction of speech and separation of Choice and Dictation object references to parameters which then can be used in python functions for further optimization and control.

**Those libraries have the focus to simplify:**

• Creation of complex grammars,

• Extraction and manipulation of phrases from the dictation.

• Separation of macros in phases of execution. Offers functionality that enables waiting for specified window, Pause for a specified time, Choice function that takes a callback function as a reference.

**5.4 IntelliJ IDEA 2016**

IntelliJ IDEA is a commercial Java IDE with advanced tools for smart code completion and refactoring, it is seen as the most efficient IDE for coding. It provides hotkeys for complete removal of reliance on a mouse (which is the most inefficient to control by voice). Simple project navigation, easy code inspection, bookmarks to jump
from one line to the other, and the list of features gets constantly increased by the myriad of updates and continuous development from the JetBrains team.

Macros triggered by voice commands were specifically designed for the IntelliJ IDE. And make use of their Hotkeys to automate common tasks like creating a new class, creating a new package and other sequence of keystrokes, resulting in time savings but if used to another IDE the user has to either update the HyperCode scripts to new hotkeys or switch to the IntelliJ IDE.

IntelliJ IDEA has a community edition which can be used for free and still keeps intact all the features provided by HyperCode.
6 STATE OF THE ART

To introduce the abilities of HyperCode and the benefits in time reduction, we present coding examples and time measurements for common programming tasks in three variations:

1. Coding using keyboard and mouse input.
2. Coding using voice input and
3. Coding using keyboard, mouse and voice input.

This is purposefully separated in three parts to serve as a contrast between the gain in performance for programmers that can use keyboards, and the speed for programmers that have to rely only on points one (1) or two (2).

Each section is separated into three (3) parts: Grammar Object, Dictation and Result.

6.1 Clarifications on time measurement

As time measurement is of prime importance to distinguish the difference and improvement HyperCode enables it requires delving deeper despite being simple.

For the first input method (keyboard and mouse) we start the time measurement from the second the user contacts with any of the devices. The user knows the task at hand and has knowledge of what he is going to do prior to typing. This will reduce any artifact introduced in the results due to the person thinking and will compare only the mechanical skills. But those vary. Mechanic skills or simply put typing speed varies from one person to the next, but the tasks are seperated to the minimum amount of code which would reduce the effect of typing speed. It is important to stress that the user using keyboard and mouse uses all hotkeys and code completions offered by the IntelliJ IDE, this is an important factor as those are to the abilities of the programmers, and the comparision should be between the times of the different input methods when those are used to the maximum.
For the second input method (voice input only) we start the time measurement from the second the user starts dictating. This measurement contains three parts of the process in which code is produced from the dictation:

1. Dictation of the command
2. Recognition of the dictation from the SR software.
3. Actions performed on the IDE.

Clearly the measurement is stopped when the last action is performed on the IDE.

For the third input method (combination of methods one and two) we start the time measurement from the second the user interacts with one of the devices or starts dictating.

6.2 The tasks

In order to portray the time measurements on all three input methods we will walk through the creation process of a Plain Old Java Object which is familiar to every Java programmer, by creating a class named Book with two (2) attributes: Author Name and number of pages. At the end of each subsection, we present the time measurements.

6.2.1 Creating the Book class

The facets of a class in Java:

- Visibility. Public or Private?
- Name of the class.
- Inheritance and
- Implementation of interfaces.

In our framework we introduced all of these possibilities in one grammar object:
6.2.1.1 Grammar Object

\[{\text{visibility}}\] \{\text{type}\} <\text{className}> \{\text{inheritance}\} <\text{inheritedClasses}>\}

Explanation of the intricacies of a grammar object:

- Square brackets [ ] mark the word as optional, meaning that the dictation is still valid when the word inside the brackets is omitted. In this example only \text{type} and \text{className} are mandatory.
- Curly Brackets { } mark the word as a Choice object. Choice objects are linked to a map of possible values. In the above grammar we have three (3) choice objects:
  - Visibility: {public, private, protected}
  - Type: {class, abstract class, enum, interface}
  - Inheritance: {extends, implements}
- Angle brackets < > mark a Dictation object that we want to extract for later manipulation, in the above example we have two (2) dictation objects: \text{className} and \text{inheritedClasses}.
  - Class Name (\text{className}) will be extracted and converted to pascal case. For e.g. dictation “paperback book” would result in PaperbackBook.
  - Inherited Classes (\text{inheritedClasses}) marks the class that is being extended or interface that is being implemented for later manipulation. See Fig. 2. that shows inherited classes in use.

6.2.1.2 Dictation

Dictation: public class book

Actions after grammar object is activated from the dictation: Our Python functions would change formatting for “book” to Pascal Case and add two curly brackets and reposition the cursor in between them with an indentation so that the code is properly formatted. See Fig. 3. for the end result.
public class Book {
}

Fig. 2. Result for dictation “public class book”

The above example will be used for the measurement of time in this task. Below we portray another example that would result in the activation of the grammar object described in this section and portrays the options and various possibilities with one well developed grammar object.

Dictation 2. public class book extends abstract book implements readable and serializable

public class Book extends AbstractBook implements Readable, Serializable

Fig. 3. Result for dictation “public class book extends abstract book implements readable and serializable”

6.2.1.3 Time Measurements

Time measurement is done only for the first dictation.

- Keyboard/Mouse: 5 seconds.
- Voice command: 4 seconds.
- Combination: 4 seconds.

Combination is just the result of voice input.

This command applies to the cases when the file named Book is present, but usually, those are added during the development.
6.2.2 Creating the new Book class

Another Grammar object that differs only by the presence of the “new” keyword, this command makes use of IntelliJ Hotkeys and the User Interface to create a new File with the same content and capabilities of the previous command.

6.2.2.1 Grammar Object

$new <fileType> <fileName>

FileType can be one of those values: enum, interface, class, annotation, singleton structure.

Filename is any continuous dictation that would be turned into pascal case and joined together.

6.2.2.2 Dictation

Dictation: “new class book”

Breaking down the actions that this command replaces:

1. Alt – Home (focuses directory of folders to enable navigation)
2. Left arrow – (move to the folder containing currently open file)
3. Alt – Insert (opens dialog for creation of new files)
4. Selects Java class from different options
5. Inserts the Dictation <fileName> in the Name field.
6. Selects Type according to the Choice {type}
7. Presses Enter
8. (Optional) If dictation contained any inheritance the code would be read and decomposed with Regular Expressions and the inheritances would be appended to the appropriate line.
In both cases, the content would be the same with the sole difference that this one ("new class book") creates the file too.

```java
public class Book {
    
}
```

Fig. 4. Result for dictation "new class book" in this case the file was created too.

```java
public enum BookGenres {
    
}
```

Fig. 5. Result for dictation "new enum book genres"

### 6.2.2.3 Time Measurements

- Keyboard/Mouse: 8 seconds.
- Voice command: 5 seconds.
- Combination: 5 seconds.

### 6.2.3 Creating primitive attributes

Primitive attributes will serve as the class fields or stated differently the properties an object can have, in this case an object of the class book.

#### 6.2.3.1 Grammar object

```
[[visibility]] {primitiveType} <variablename>
```

Visibility can be one of the following values: pubic, private, protected, or it can be omitted entirely as shown by the presence of the square brackets.

PrimitiveType can be one of the following values: byte, short, int, long, float, double, char, boolean.

Filename is any continuous dictation that would be turned into camel case and joined together.
6.2.3.2 Dictation

**Dictation:** “private int number of pages”

```c
private int numberOfPages;
```

Fig. 6. Result for dictation “Private int number of pages”

**Dictation 2.** Public boolean is out of print

```c
public boolean isOutOfPrint;
```

Fig. 7. Result for dictation "public boolean is out of print"

6.2.3.3 Time Measurements

Time measurements:

- Keyboard/Mouse: 6 seconds.
- Voice command: 4 seconds.
- Combination: 4 seconds.

6.2.4 Creating object attributes

6.2.4.1 Grammar object

```c
[[visibility]] <Type> instance <variablename>
```

Visibility can be one of the following values: public, private, protected, or it can be omitted entirely as shown by the presence of the square brackets.

Type can be a sequence of words which will be joined together and formatted to pascal case. This sequence of words is interrupted by the “instance” keyword and it marks the beginning of the next dictation object.

Variablename can be any sequence of words that will be formatted in camel case.
6.2.4.2 Dictation

Sample dictation that activates this Grammar Object:

**Dictation:** “*private string instance author name*”.

```java
1 private String authorName;
```

Fig. 8. Result for dictation “private string instance author”

**Dictation 2:** “*private big decimal instance price of cost*”

```java
1 private BigDecimal priceOfCost;
```

Fig. 9. Result for dictation “private big decimal instance price of cost”

6.2.4.3 Time Measurements

- Keyboard/Mouse: 6 seconds.
- Voice command: 4 seconds.
- Combination: 4 seconds.

6.2.5 Creating the class constructor

6.2.5.1 Grammar object

Add constructor.

6.2.5.2 Dictation

Dictation: “*Add constructor*”.

```java
1 public Book(String authorName, int numberOfPages) 
2 {
3  this.authorName = authorName;
4  this.numberOfPages = numberOfPages;
5 }
```

Fig. 10. Result for dictation “Add constructor” while inside a class Book with two attributes authorname and number of pages
Another example of this grammar object while we are present in a class Player who has two attributes name and score. Dictation doesn’t change but the result does:

```java
public Player(String name, int score)
{
    this.name = name;
    this.score = score;
}
```

Fig. 11. Result for dictation “Add constructor” while inside a class Player with two attributes name and score

Those special commands are referred as Context Aware Commands in the continuation of this paper.

**Context Aware Commands**

Hypercode contains Context Aware Commands that produce different results according to the contents of the file currently in.

In the above example when we dictated “Add constructor” while being present in the file with the code as shown in Fig. 12.

```java
public class Book
{
    private String authorName;
    private int numberOfPages;
}
```

Fig. 12. Code of class Book

HyperCode functions read the contents of the file by selecting everything on the page, and copying them on the clipboard, the function continues to extract information using regular expressions from the contents of the clipboard and finally insert the extracted information into the output and produce the result shown earlier in Fig. 10.

Similarly in the second example with the class Player with two attributes, the class name and the attribute types and name were extracted and the constructor was formed and printed into the IDE.
Alternative Commands for constructor creation:

- Add empty constructor
- Add constructor initialize author name and number of pages (for partial attribute initialization)

6.2.5.3 Time measurements

- Keyboard/Mouse: 5 seconds.
- Voice command: 5 seconds.
- Combination: 5 seconds.

This is a case we would have expected a much better result, Add constructor seems an easy command that should have produced a much faster result then the alternative of typing. But this is the case when IDEs have already automated features that can be used by users that know the shortcuts.

6.2.6 Creating getter and setter methods

Because of the linear and rigid structure of these methods, there are prepared templates in IntelliJ, our command makes use of those templates, using UI and hotkeys.

6.2.6.1 Grammar Object

Get <attributeName>.

Attributename can be any sequence of words that will be formatted into Pascal Case, and get in lower case will be appended in front of it. It would be the same process for a setter method.

6.2.6.2 Dictation

Dictation: Get author name.

```java
public String getAuthorName() {
    return authorName;
}
```

Fig. 13. Result for dictation “Get author name”
6.2.6.3 Time measurements

- Keyboard/Mouse: 2 seconds.
- Voice command: 3 seconds.
- Combination: 2 seconds.

Alternative voice commands for getter and setter creation:

- Add getters and setters (uses context awareness)

6.2.7 Generating the Equals method

To create the equals method, we prepared a template that gets filled using context awareness with information and cursor is positioned after return so the user can decide how he wants to evaluate the objects for equality.

6.2.7.1 Grammar object

Generate equals method.

6.2.7.2 Dictation

Dictation: Generate equals method.

```java
public boolean equals(Object object) {
    if (object == this)
        return true;
    if (!(object instanceof Book))
        return false;
    Book book = (Book) object;
    return
}
```

Fig. 14. Result for dictation “Generate equals method”
6.2.7.3 Finishing the Equals method

In order to create the equals method we use multiple grammar objects. Below we have the total sequence of commands to create the equals method for the class Book.

Sequence of commands:

1. Camel author name.
2. Object options. (see: Fig. 9.)
3. Select option 1.
5. Object options.
6. Select option 2. (Option that is method getAuthorName)
7. Go end.
8. Semicolon.

Fig. 15. IntelliJ’s UI for selecting options available for an object.

```
9   return authorName.equals(book.getAuthorName());
```

Fig. 16. Result of the sequence of commands in section 6.2.7.3
6.2.7.4 Time measurements

- Keyboard/Mouse: 15 seconds.
- Voice command: 29 seconds.
- Combination: 12 seconds.

In this task we can see that when the commands contain a lot of symbols the time to insert information into the editor dramatically slows down due to inability to compose those commands with continuous speech ability.

Another very important result from this measurement is that the combination has a smaller time then both the other input methods, this is due to the fact that dictation of Generate equals method produced a large part of the code and finishing the last part with the different symbols was much faster with the keyboard. This is one of the cases where the combination provides great speed up.

6.2.8 Creating the toString method

6.2.8.1 Grammar objects and Dictations

Sequence of commands:

1. Dictation: `public method string instance toString`.
2. Camel author name
3. Plus sign
4. Inside quotes.
5. Colon
6. Go end
7. Plus sign
8. Camel number of pages.

```java
public String toString()
{
    return authorName + ":" + numberOfPages;
}
```

Fig. 17. Result of the sequence of commands in section 6.2.8.1
6.2.8.2 Time measurements

- Keyboard/Mouse: 18 seconds.
- Voice command: 30 seconds.
- Combination: 10 seconds.

Just as on the previous example we notice that Voice input alone slows down immensely when we get to the point of different and multiple symbol dictation where each requires a pause, hence continuous speech recognition capabilities cannot be used to the maximum.

And again we can see that in those cases where we have a large code that is always the same it can be automated and then we use the keyboard where we have heavy different symbol usages.
7 CONCLUSION

According to the presented examples, we conclude that with HyperCode voice input can be used as viable solution for people with ULD, up to the point of replacing the need for a keyboard entirely. Additionally, by leveraging the hot-keys offered by IntelliJ IDEA our framework enables a programmer to create new classes and files to form complex structured programs.

Table 1 shows the total time for the examples in section three (3) per input method. A combination of keyboard input and coding by voice using HyperCode makes a programmer more efficient at inserting information into the computer. In this sample of basic exercises the combination was faster by 19 seconds in contrast to keyboard and mice input. Meanwhile voice input only is slower but at an acceptable speed.

<table>
<thead>
<tr>
<th>Input method</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard/Mouse</td>
<td>65 seconds</td>
</tr>
<tr>
<td>Voice input</td>
<td>84 seconds</td>
</tr>
<tr>
<td>Combination</td>
<td>46 seconds</td>
</tr>
</tbody>
</table>

In the end all three goals were achieved, speech recognition increases overall efficiency, hotkeys were automated and enables control of the user interface and by doing so enables people with disabilities to code.

In order to increase speech recognition accuracy HyperCode could be used in conjunction with a module that extracts and stores project data into a database. That can be used to increase the priority of class names, attributes and methods during speech recognition, leading to fewer speech recognition errors, and this is just one of the improvements other options would be to make the python scripts update themselves as requested from the user in order to create more personalized usage according to the needs of the application being developed.
APPENDIX

Additional Features

In this subsection we present some voice commands that are fundamental to the functionalities of HyperCode:

- Navigation functionality is mapped with Voice Commands (e.g. “Move up 5 lines”, “Switch to <className>” etc.)
- Go to {Top, Bottom} of file.
- Page up, Page down.
- Move <n> lines in {direction}.
- Move <n> words on the {direction}.
- Go to method declaration.
- Run [this] job.
- Code editing with context awareness (e.g. “Make class abstract”, “make method private”, “implement interface <interfaceName>” etc.)
- Git functionalities (“Commit file”, “Pull from branch”, “Push changes”)
- Delete block content(context aware command, looks for brackets and deletes its contents).
- Add annotation <annotationName> to {attribute, method, class} <name>.
- Add parameter (adds separation with comma automatically).
REFERENCES

Bibliography


