Bilkent University

Department of Computer Engineering

Senior Design Project

Project Short Name: Clerk

Final Report

Ahmet Malal, Ensar Kaya, Faruk Şimşekli, Muhammed Salih Altun, Samet Demir

Supervisor: Prof. Uğur Doğrusöz

Jury Members: Prof. Varol Akman and Prof. Çiğdem Gündüz Demir

Innovation Expert: Mehmet Surav

Final Report
May 27, 2020
This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.

Contents

1 Introduction

Microsoft Word is one of the most popular word-processing programs around the world. It is used primarily to create various types of documents that you can print and publish, such as books, papers, and reports. When you open a document in Microsoft Word, it can be edited using various features that Word provides. Currently, these features are accessible with the use of some input devices, such as mouse and keyboard.

Microsoft allows developers from around the world to develop applications that will extend the functionality of Word. These applications are called "Word add-ins". A Word add-in is an application which is essentially embedded inside of Word and can be launched from within a running Word instance. We want to build our application as an add-in for Word because it is the most popular text editor there is and it fits best with the nature of the work we want to do, which is to extend the conventional text-editor functionalities and make text-editing more accessible for people.

We have built an add-in for Word that will allow users to utilize some of Word's features along with some additional external features using voice commands. The application also has a local server component, along with the add-in itself.

In this report we first present the final set of requirements for our application, then we talk about the design, architecture, and implementation in detail. Following that we talk about testing and maintenance plans and other elements. We end the report with our conclusions about the project and details of future work. A user's manual is also provided in the Appendix.

2 Requirement Details

2.1 Functional Requirements

Receiving Voice Input

- The user will be able to give voice input to the application using any voice input device.

Switching Modes

- Clerk will have three modes, i.e. states.
- In sleep mode voice input will be taken but not analyzed unless it is a special, mode switching sequence or a button to switch modes is pressed.
- In speech mode Clerk will take voice input and convert it into text, i.e. it will let the user dictate what they want to type. Commands will not be recognized in this mode.
- In command mode, only commands will be recognized and applied, speech won't be converted to text.
- The application will start in sleep mode. The user can switch to speech mode by using the "Wake Up" command or use a button that will be provided in the user interface.

- When the application is in speech mode, the user will be able to switch to command mode by pressing the space bar.

Converting Voice Input to Text Commands

- Clerk will receive voice commands in command mode.
- The input received will be converted to text to be parsed.
- The text input will be parsed to understand the contents. The application will figure out which functionality the user wants to use, such as typing, copying, pasting, or saving the file. Then the command will be executed.

Dictating Text with Voice Input

- In speech mode, the user will be able to dictate the words getting typed into the document with their voice.
- Clerk will recognize punctuation alongside words.

Read-back Functionality for Error Checking

- The user will be able to request a read-back of a word, sentence, or paragraph from where the cursor currently is. This means that the program will convert the text in the document to speech and read it to the user.

Saving the File

- Clerk will provide a command for saving the current file.

Document Navigation

- The application will provide satisfactory navigation inside the document. It will have commands to be able to navigate to certain paragraphs, sentences, and words. It will also provide the functionality to search for phrases within a document, and navigate through occurrences of those phrases.

Finding Appearances of a Text

- The application will provide the functionality to search for phrases within a document, and navigate through occurrences of those phrases.

Selecting a Range of Text

- The user will be able to select and highlight a range of text using voice commands. They can select a specific word, sentence, or paragraph, as well as selecting all text within some range specified by the user.

Reading a Range of Text to the User

- The user will be able to request Clerk to read them a range of text.
- The application will read words to the user by converting the text in the document to speech.

Deleting a Range of Text

- The user will be able to delete the currently highlighted range of text by giving voice commands.
- The user will be able to delete any range of text.
- The user will be able to delete the word, sentence, or paragraph that the cursor is currently on by giving voice commands.
- The user will be able to delete the last sentence of the document or a certain paragraph by giving voice commands.
- The user will be able to delete all text and objects.

Copying and Pasting Text from/to Clipboard

- Clerk will provide commands for copying and pasting text in command mode.
- The user will be able to cut or copy the selected range of text, as well as request the last word, sentence, or paragraph to be cut or copied using voice commands.
- The user will be able to paste the portion of the text in the Clipboard into the document wherever the user wishes.

Changing Font

- Clerk will provide commands to change the font, font size, color; set text to be bold, italic, underlined.
- The user can format their text in any way they want, and can also clear all formatting from a range of text.

Providing a Definition for Words

- The user will be able to find the definition of a range of words s/he has selected. If the user does not select any word Clerk will open the definition of the current word where the cursor is.
- Clerk will find definitions in the Cambridge Dictionary website.

Searching in the Web

- If the selected range or the word, on which the current location of the cursor is, is a website link, the user will be able to go to the related website on the Web.
- If the selected range or the word, on which current location of the cursor is, is not a website link, the user will be able to google it on a browser.

Repeating the Last Command

- The user will be able to repeat the last successfully parsed command by saying "repeat".
- The user will be able to repeat it as many times as they want by saying, for instance, "repeat 3 times".

Spelling a Word

- The user will be able to ask for a spelling of a word where the cursor is. This will help find typos if there is one.

Finding Similar Words for a Selected Word

- Clerk will provide similar words for a word that the user has selected.

2.2 Non-functional Requirements

Reliability

- The application must be able to convert voice to text with acceptable accuracy. Errors and misunderstandings must happen rarely. When they do happen, the user needs to be alerted appropriately.
- The application must not randomly crash. It must be able to recover from most errors. It must be able to communicate the errors and the reasons they occurred to the user as much as possible.

Efficiency

- The application must not take more than 5 seconds to process and respond to any voice command. This requirement includes a speech to text delay and parsing. Longer delays could turn the user away from the use of the application.

Extensibility

- It must be easy to develop new features and add new functionality to the application. This requires the application to be sufficiently modular, which will be achieved by using Object Oriented Programming paradigms and appropriate design patterns.

Compatibility

- The application will be able to work on multiple platforms that can run Word such as Windows, Mac, iPad, and web browser. This feature is provided by Microsoft themselves [1].

Usability

- Once the application is launched, the user should be able to perform all the functionality using only voice commands from an audio input device. There should not be a need to use any other devices. This is especially important since our target audiences include people who aren't able to use these devices.
- Despite working only with voice commands, Clerk will still provide a user interface to its users. Some functionality such as setting the mode can also be used from the user interface.

Error-Handling

- The application must handle errors as much as possible and provide acceptable feedback to the user about them.

Licensing

- The application will be built using Microsoft Developer Licenses to test in Word.
- Licensing for the third-party libraries will be adhered to.

2.3 Pseudo Requirements

- Clerk will be developed as a Microsoft Word Add-in application.
- Clerk will work only when connected to the Internet.
- English language will be supported.
- Javascript, Node.js, HTML, and CSS will be used to develop the add-in.
- Git and GitHub will be used for version control and collaboration.
- Word Javascript API will be used in the development of the add-in [2].
- Common API from Office365 will be used to control the objects and metadata in Word documents [3].
- The application will work on the following platforms: Word 2013 or later on Windows, Word on the web, Word 2016 or later on Mac, and Word on iPad.
- The application will be developed under Object-Oriented Programming paradigms.

- Clerk will be an open-source Microsoft Word Add-in that will be available to use for free to all Word users.
- Licenses for third-party APIs and libraries will be checked before usage. Free and open-source libraries and APIs will be used.
- Personal information and private data will not be shared with any third parties.
- Microsoft Word already encrypts the user data, which is accessible only after authentication.
- To improve the usability of the application, feedback from the users will be considered. The application will be updated by the developers according to feedback taken from the users.
- The consent of the user is required since we will take their voice input and process it.
- The application will not store the voice input content in any way. So, there is no way we can share data with third parties.

3 Final Architecture and Design Details

3.1 Final Architecture

3.1.1 Client-Server Architecture

Clerk uses the microphone and speaker to interact with the user. Therefore, Clerk needs Text-to-Speech (TTS) as well as Speech-to-Text (STT) functionalities. The architecture of STT and TTS is Neural Networks. Besides, Clerk is an application running on en Edge browser. Therefore, the language of Clerk should be one of the languages compatible with HTML. As a design decision, we have decided to use JavaScript in our application. It can be stated that JavaScript is not a useful language for Neural Network structure. Thus, we decided not to use any JavaScript library due to efficiency concerns. Besides, we discovered that `Google Cloud Text to Speech API` is a highly accurate and convenient library and compatible with `Node.js`. Since `Node.js` is not compatible with the Edge browser, we had to write it as a different program. since `Node.js` will only be used to send requests and get the corresponding data such as transcription, it acts as a server. And, since our add-in is the requester, it acts as a client. Therefore, we have decided to have Client-Server architecture. And, the data-flow between Server-side and Client-side is managed by the sockets, which provides bilateral data-flow. This architecture is utilized for the implementation of Copy and Paste commands, Lex-Yacc structure, and especially for TTS and SST.

3.1.2 MVC Architecture of Client Side

Since we use JavaScript for Client-side, which is an Interpreted Language and is not designed for Object-Oriented Programming (OOP), we have tried our best to use JavaScript as an OOP language. Therefore, the architecture of Client-side is in between them.

As a design decision, we have used MVC Architecture for the Client-side. The Controller side is to communicate with the server-side and take the transcription, then decide which Command to be executed and send it to the Model side to execute the command. The Model side is to execute the commands detected by the Controller. By the definition of the Model side, it does not interact with the classes in itself and is used when only when Controller triggers. And, the View side is for User Interface and interacts with Controller.

3.2 Design Details

3.2.1 General Flow

The general flow of Clerk can be explained in four major steps. First, it takes speech input from the user using a recorder on the server-side which runs locally. Secondly, it converts the speech to a transcription using a Speech to Text API. After that, it detects the expected behavior of the transcription according to the current mode of the system. If the Clerk is in sleep mode, the system waits for a "Wake up", or if the Clerk is in speech mode, system types whatever the transcription and finally if Clerk is in the command mode, parameters and type of the command are detected from the given transcription using a parser which uses a lexical analyzer and parser. And the last step of command mode is to execute the detected command on the client-side. A sequence diagram is given below for better understanding of general workflow which have a simple scenario from the opening of the Clerk to printing out "Hello world" words on Word.

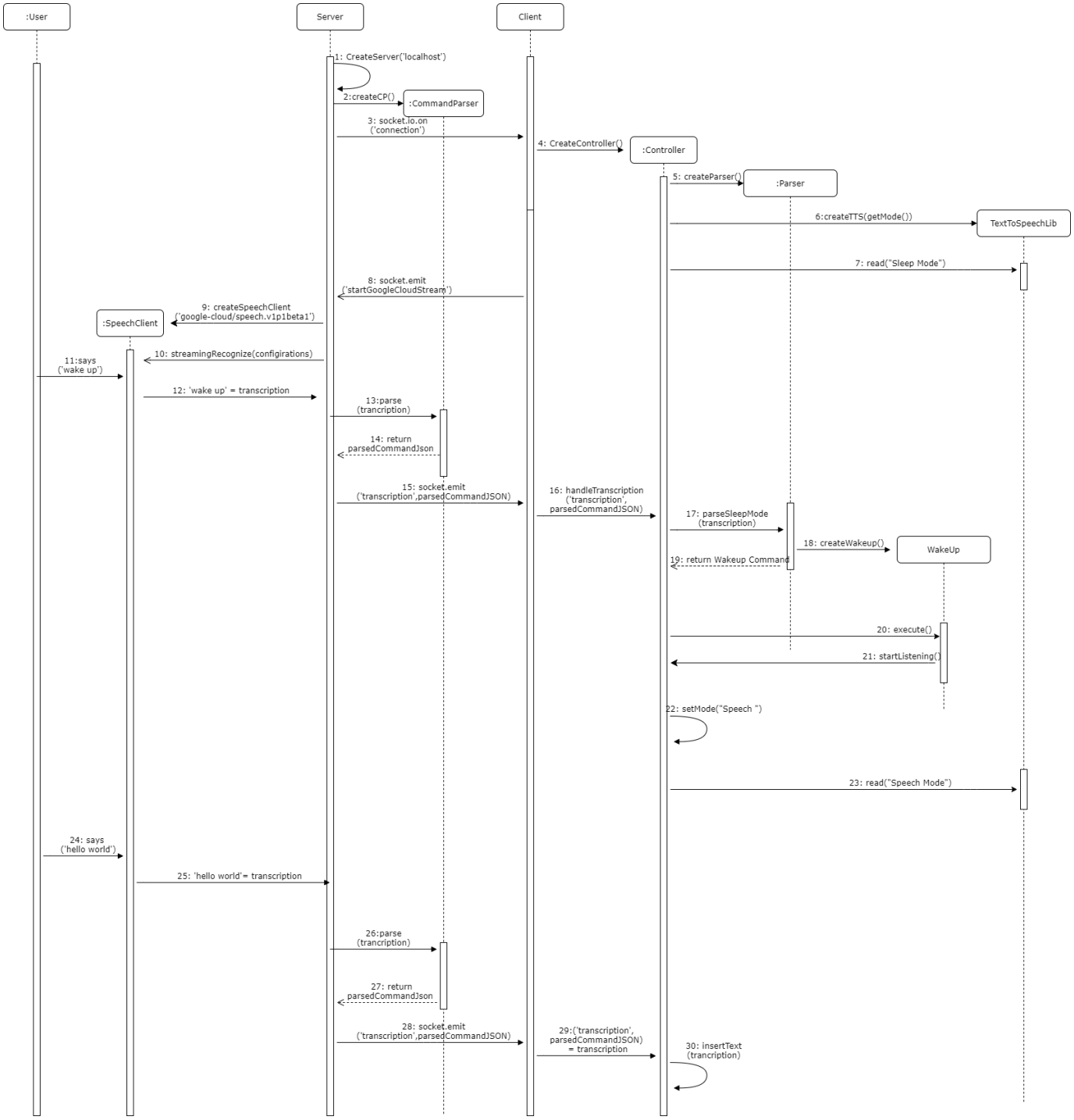

Figure 1: A sequence diagram of our project

3.2.2 Client

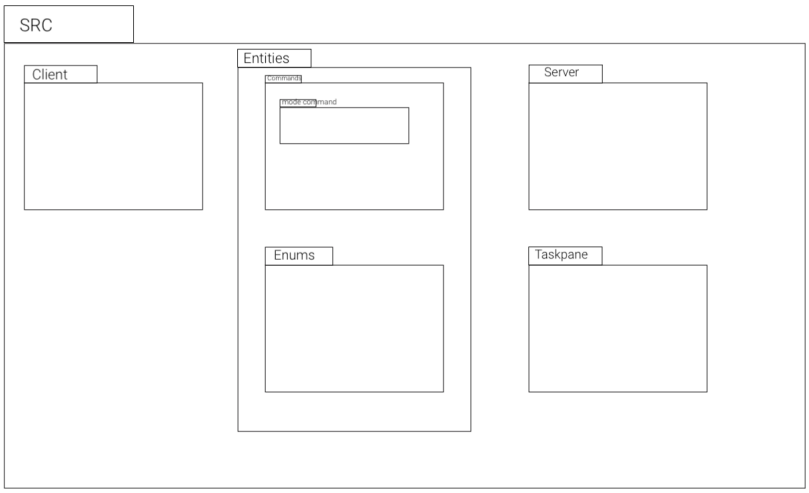

Figure 2: The package diagram of our project

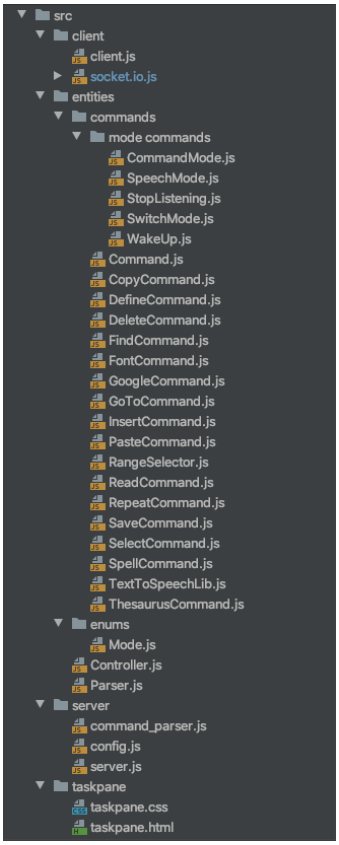

Figure 3: The screenshot of our package tree

Controller of MVC Architecture: The structure of the Controller of MVC Architecture is as follows. To communicate with the server-side we have a class called `Client`. This class just takes the transcription and `parseTree` (JSON object) from the server and sends it to `Controller` class. Then, `Controller` class is to handle the transcription by considering the current mode of Clerk.

If the mode is `SPEECH` mode, it just inserts the text to Word.

If the mode is `SLEEP` mode, it checks if the `parseTree` belongs to a wake-up command or not, and switches the mode to `SPEECH` mode.

If the mode is `COMMAND` mode, then it sends the `parseTree` to `Parser`. `Parser` detects which command type the parseTree belongs to and returns an instance of the detected command to `Controller`. Then, `Controller` calls the `execute()` method of the command object detected by `Parser`.

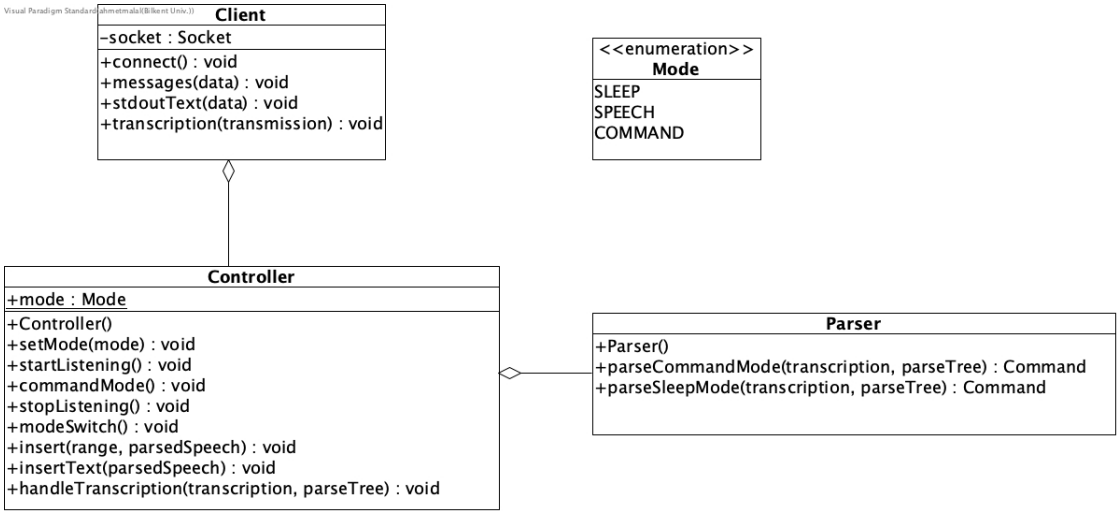

Figure 4: The class diagram of Client side

Client

It is a bridge between `Controller` class and server side. It receives the transcription from server side and sends it to Controller.

Class	**Client**
Attributes	
`socket`	It is a socket connected to server side. It is used to send/recieve data to/from server side.
`controller`	It is an instance of `Controller`
Methods	
`transcription(transmission)`	It takes transmission JSON Object that wraps transcription and parseTree and send it to Controller to execute it
`stdoutText(data)`	It prints the transcription on UI.
`messages(data)`	It prints the data to console. It is a log function for debugging purposes.
`connect()`	It starts a connection with the server side
`stopRecording()`	It stops the recording and ends streaming

Parser

This class manages to parse the JSON object to the corresponding Command object.

Class	**Parser**
Methods	
`parseCommandMode(transcription, parseTree)`	It parses the given `parseTree` and returns `Command` object
`parseSleepMode(transcription, parseTree)`	It checks if the `parseTree` is a `WakeUp` command or not. If so, returns `WakeUp` command instance.

Controller

This class is responsible for managing the transcription coming from Client.

Class	Controller
Attributes	
`mode`	It keeps the state information of Clerk system. It can be `speechMode`, `commandMode` or `sleepMode`.
Methods	
`setMode(mode)`	It is a set method of mode
`startListening()`	It changes the mode to `Speech` mode
`commandMode()`	It changes the mode to `Command` mode
`modeSwitch()`	It switches the mode between `Command` mode and `Speech` mode
`stopListening()`	It changes the mode to `Sleep` mode
`getModeString()`	It returns the name of the current mode
`insert(range, parsedSpeech)`	It inserts `parsedSpeech` by replacing the given `range`. It is used to insert the text in `Speech` mode.
`insertText(parsedSpeech)`	It inserts `parsedSpeech` by replacing the selected Range. It is used to insert the text in `Speech` mode.
`handleTranscription(transcription, parseTree)`	It finds the corresponding command object and calls its `execute()` method

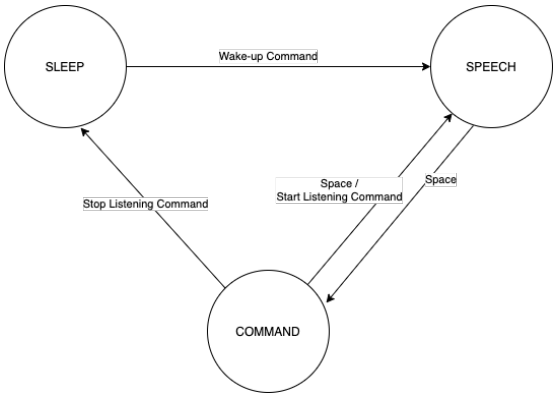

Figure 5: The state diagram of `mode`

Clerk has `SLEEP`, `SPEECH`, and `COMMAND` modes that can be considered as the states of the program. The user is able to switch the state from `SLEEP` to `SPEECH` mode by calling `WakeUpCommand`. The user is able to switch the state of the program from `SPEECH` mode to the `COMMAND` mode or from `COMMAND` mode to `SPEECH` mode by pressing the `space` button. In addition to this, the user can change the state of the program from `COMMAND` mode to `SPEECH` mode by calling `StartListeningCommand`. By calling `StopListeningCommand`, the state of the program changes from `COMMAND` to `SLEEP` mode. The state diagram of the program is above.

Figure 6: The class diagram of Model classes

Model of MVC Architecture:

Command

It is the superclass of other command classes. Since we have designed it to be an abstract class, along the way we use the children of this class to execute different commands. You can see the attributes and methods of Command class in the table below.

Class	**Command**
Attributes	
`parseTree`	It stores the parser in JSON format
Methods	
`Command(parseTree)`	It is the constructor method which takes the parse tree

TextToSpeechLib

The program can read a written text in the document in several places such as a whole paragraph or when the state changes. When needed, we use this class to carry out the operation. A specific collection of words is sent to our socket, the text is converted to speech in mp3 file format and it is sent back. We can listen to the speech. Also, we can pause the speech whenever asked. You can see the attributes and methods of the TextToSpeechLib class in the table below.

Class	**TextToSpeechLib**
Attributes	
`audio`	It stores the audio which is in .mp3 format
Methods	
`read(text)`	It takes a text and sends to the socket
`pause()`	It pauses the audio if the audio is open

SpellCommand

Since we work with the SpeechToText library, its accuracy is a bottleneck for our program. So, in case of inserting difficult words that are hard to be understood by the library, into the document, the user may want to spell some specific group of words for correction purposes. Therefore, this class splits the selected word, sentence, or paragraph into a character array and sends the character array to TextToSpeechLib to be voiced. You can see the attributes and methods of SpellCommand class in the table below.

Class	**SpellCommand**
Methods	
`SpellCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes a spell command specified by the parse tree that contains related information

RangeSelector

Class	RangeSelector
Methods	
`getCurrentWord()`	It returns the range of the current word that is where the cursor is
`getCurrentParagraph()`	It returns the range of the current paragraph that is where the cursor is
`getCurrentSentence()`	It returns the range of the current sentence that is where the cursor is
`getAllDocument()`	It returns the range of the whole content in the document
`getIndexes(indexClause)`	It returns indices for paragraph, sentence, and word using the given parameter
`getSentenceFromCurrentParagraph(index)`	It returns the range of the sentence at a given index in the current paragraph
`getWordFromCurrentSentence(index)`	It returns the range of the word at a given index in the current sentence
`getParagraph(index)`	It returns the range of a paragraph at a given index
`getNextParagraph()`	It returns the range of the next paragraph from where the cursor is
`getPreviousParagraph()`	It returns the range of the previous paragraph from where the cursor is
`getNextSentence()`	It returns the range of the next sentence from where the cursor is
`getPreviousSentence()`	It returns the range of the previous sentence from where the cursor is
`getNextWord()`	It returns the range of the next word from where the cursor is
`getPreviousWord()`	It returns the range of the previous word from where the cursor is
`getRangeFromTo(indexes1,indexes2)`	It returns range between two indexes arrays. These can be any type of index clause such as second word of the next paragraph
`getWordFromSentence(index1,index2)`	It returns the range of the word at a given index1 in the sentence at a given index2
`getSentenceFromParagrah(index1,index2)`	It returns the range of the sentence at a given index1 in the paragraph at a given index2
`getWordSentenceParagraph(index1,index2,index3)`	It returns the range of the word at a given index1 in the sentence at a given index2 in the paragraph at a given index3

For some specific commands such as delete, select, first we should be able to find the range needed. We designed this class to provide a range for very flexible inputs (see User Manual). A simple example would be deleting the second word of the previous sentence or a much larger range. After we find the range asked, a command is being carried out. You can see the attributes and methods of the `RangeSelector` class in the table above.

CopyCommand

For some reason, the user may want to use some parts of the document again. In this case, this class copies the selected word, sentence, or paragraph into the clipboard in order to be used later. You can see the attributes and methods of CopyCommand class in the table below.

Class	**CopyCommand**
Methods	
`CopyCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes a copy command specified by the parse tree that contains related information

SelectCommand

We need to be able to select a piece of the document before some commands such as read that text is being read to the user. This class allows the user to highlight that piece using the given parameters in the parse tree that comes with the constructor. Since our RangeSelector is quite flexible, we can highlight any portion of the document easily. You can see the attributes and methods of SelectCommand class in the table below.

Class	**SelectCommand**
Methods	
`SelectCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes a select command specified by the parse tree that contains related information
`selectAllDocument()`	It selects everyting written in the document

ReadCommand

This class is responsible for sending the selected word, sentence, or paragraphs to TextToSpeechLib class. You can see the attributes and methods of ReadCommand class in the table below.

Class	**ReadCommand**
Methods	
`ReadCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes a read command specified by the parse tree that contains related information

GoToCommand

This class allows the user to move the cursor anywhere in the file so that the user can complete another command such as insertion of text into the document. It changes our view of the document. You can see the attributes and methods of GoToCommand class in the table below.

Class	**GoToCommand**
Methods	
`GoToCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes a goTo command specified by the parse tree that contains related information

DeleteCommand

The user may want to delete some parts of the document when it is not needed. This class allows the user to remove the selected word, sentence, or paragraph from the document. You can see the attributes and methods of DeleteCommand class in the table below.

Class	**DeleteCommand**
Methods	
`DeleteCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes a delete command specified by the parse tree that contains related information
`deleteAllDocument()`	It deletes everything written in the document
`deleteSelection()`	It deletes the selected range in the document

GoogleCommand

The user may frequently come across website links in the document and want to open the related website in a browser. This class provides this functionality to the user. You can see the attributes and methods of GoogleCommand class in the table below.

Class	**GoogleCommand**
Methods	
`GoogleCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes a google command specified by the parse tree that contains related information

FontCommand

It is quite common that the user may want to change the font of a selected piece of text in the document. In this class the user can change different types of font-related data such as size and color of the text, and whether being italic, bold and underline. This class allows the user to change any of the font types with ease. You can see the attributes and methods of FontCommand class in the table below.

Class	FontCommand
Attributes	
`size`	It stores the size of font
`color`	It stores the color of font
`italic`	It stores the information about whether font is italic or non-italic
`bold`	It stores the information about whether font is bold or not bold
`underline`	It stores the information about whether font is underlined or not
Methods	
`FontCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes a font command specified by the parse tree that contains related information

FindCommand

Many times the user wants to find the appearances of a group of words in the document and examine them. This class provides this functionality. And with the methods of this class, the user can proceed to the next appearance of the text or go back to the previous one. You can see the attributes and methods of FindCommand class in the table below.

Class	FindCommand
Attributes	
`searchString`	It stores the search text
`searchResults`	It stores the results in a Range array
`searchResultsIndex`	It stores the current search result index
Methods	
`FindCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes a find command specified by the parse tree that contains related information.
`find(str)`	It is a set method of mode
`next()`	It finds the next range of result of `searchString`
`previous()`	It finds the previous range of result of `searchString`

DefineCommand

The user may frequently come across words of which s/he does not know the meaning and wants to look up in the dictionary. This class provides this functionality to the user. It finds and opens the meaning of a selected word in the Cambridge dictionary on a browser. You can see the attributes and methods of GoogleCommand class in the table below.

Class	DefineCommand
Methods	
`DefineCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes a define command specified by the parse tree that contains related information

InsertCommand

The core requirement of the program is to insert something into the document. Therefore, along the way, the user would like to insert a new line, new paragraph, new page or any text s/he wishes. The user can go to a specific position in the document that is moving the cursor using GoToCommand class and insert the text there. This class provides this functionality to the user. You can see the attributes and methods of InsertCommand class in the table below.

Class	**InsertCommand**
Methods	
`InsertCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes an insert command specified by the parse tree that contains related information

PasteCommand

Copy-Paste is one of the operations that is carried out mostly in the MS Word. To do this, first, the user should have copied a piece of text in the document that s/he wishes paste using CopyCommand. And this class allows the user to paste the previously copied text into the position where the cursor is, also that can be changed using another command: GoToCommand. You can see the attributes and methods of PasteCommand class in the table below.

Class	**PasteCommand**
Methods	
`PasteCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes a paste command specified by the parse tree that contains related information

RepeatCommand

The user may want to execute the last command that is carried out again, even for as many time as possible. This class provides this functionality for the user. You can see the attributes and methods of RepeatCommand class in the table below.

Class	**RepeatCommand**
Attributes	
`lastCommand`	It stores the last successfully executed command
`lastCommandStr`	It stores the last successfully executed command transcription
Methods	
`RepeatCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes the `lastCommand`

SaveCommand

This class allows user to save the document not to lost information. You can see the attributes and methods of SaveCommand class in the table below.

Class	**SaveCommand**
https://www.overleaf.com/project/5ec6d30e49fa5c0001c334ef **Methods**	
`SaveCommand(parseTree)`	It is the constructor method w
`execute()`	It executes a save command s contains related information

TheasurusCommand

Especially students may have a need to find similar words and chunks for a specific word not to overuse that word. This class provides this functionality for the user. After selecting the word s/he wishes, the program opens related webpage in a browser that shows similar words for it. You can see the attributes and methods of TheasurusCommand class in the table below.

Class	**TheasurusCommand**
Methods	
`TheasurusCommand(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It executes a theasurus command specified by the parse tree that contains related information

CommandMode

This class changes the state of the program from `SPEECH` to `COMMAND` so that the user will be able to execute commands afterward. This rule out the confusion whether to insert into the document or execute as a command. You can see the attributes and methods of CommandMode class in the table below.

Class	**CommandMode**
Methods	
`CommandMode(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It sets the state of the program to `COMMAND`

StopListening

This class changes the state of the program from `COMMAND` to `SLEEP` so that the user will be able to execute commands afterward. Except for WakeUp command, it neither executes command nor inserts text. You can see the attributes and methods of the StopListening class in the table below.

Class	**StopListening**
Methods	
`StopMode(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It sets the state of the program to `SLEEP`

SpeechMode

This class changes the state of the program from `COMMAND` to `SPEECH` so that the user will be able to insert text into the document afterwards. This rule out the confusion whether to insert into document or execute as a command. You can see the attributes and methods of SpeechMode class in the table below.

Class	**SpeechMode**
Methods	
`SpeechMode(parseTree)`	It is the constructor method which takes the parse tree
`execute()`	It sets the state of the program to `SPEECH`

SwitchMode

This class is responsible for switching the state of the program between `SPEECH` and `COMMAND` as response to key press that is determined. You can see the attributes and methods of SwitchMode class in the table below.

Class	**SwitchMode**
Methods	
`SwitchMode(parseTree)`	It is the constructor method which takes the parse tree.
`execute()`	It switches the state of the program between `SPEECH` and `COMMAND` as response to key press

WakeUp

This class changes the state of the program from `SLEEP` to `SPEECH` so that the user will be able to insert text into the document afterward. This rule out the confusion whether to insert into the document or execute as a command. You can see the attributes and methods of the SpeechMode class in the table below.

Class	WakeUp
Methods	
`WakeUp(parseTree)`	It is the constructor method which takes the parse tree.
`execute()`	It sets the state of the program to `SPEECH`

3.2.3 Server

Since the server's design is not object-oriented, it will be described as it's major components or modules.

Unlike the traditional Client-Server model, the Clerk server has more of a helper component role in that it is used only locally, to take speech input from the user and send it, as a sound signal, to the Speech to Text API.

This design decision results from the fact that even though the Word application runs a browser inside, EdgeHTML version 18, to host the add-in, it doesn't permit to use the microphone. Since we couldn't solve this problem, we had to figure out another way to listen to the user. The way we managed to do that was by using a local server that runs Node.js and listens to the user in the background and sends information to and receives information from the client via the use of sockets.

The server has five major modules.

- **Recorder:** Records sound from an input device. The sound signal is piped to an outgoing data stream that connects to the Speech to Text service.
- **Speech to Text Service:** Takes sound signal as input and returns the most likely transcription as text.
- **Command Parser:** Takes the transcription as input and returns a parse tree based on predefined rules of Clerk commands which is a language defined by a context-free grammar.
- **Networking:** Uses sockets to communicate with the Clerk client application. Sends messages to and receives messages from the client and acts accordingly. It is used in sending transcriptions, receiving text for Text to Speech, and receiving messages for copy and paste operations.
- **Text to Speech Service:** Takes a message in text form that is to be communicated to the user. Returns a playable sound file that is written to disk to be played and then deleted.

Recorder

The recorder module is used to record sound from an input device. It buffers the sound data until a period of silence is hit and then pipes it to the outgoing data stream which is used by Speech to Text module to communicate with the Speech to Text service. The recorder uses the default input device of the operating system at the time of launch. There is only a single instance of recorder available in the application at all points.

Speech To Text

The Speech to Text module is used to get a transcription for the recorded sound. Clerk utilizes external APIs that use Neural Networks that have been trained on many hours of data to transcribe sound. This means that there will always be certain errors in transcription, though the error rate will be low. The Speech to Text module will always return the transcription for which it's internal model has the highest confidence value.

Command Parser

From the Programming Languages course, we know that Lex and Yacc are used in the parsing stage of a compiler. It parses the given text input and returns whether the input is rejected for that language or not. Also, we know that we can obtain the parse tree of the parsed text in Yacc. Besides, for the command parsing step of Clerk general flow we needed a tool or algorithm to parse the given text to such a format that will ease the execution step. We could have used Natural Language Processing (NLP) for this purpose. However, it would not be 100% accurate and we needed to have a dataset for each command to train the algorithm. We know that the accuracy is important in our application since it would be annoying if we executed a different command than what the user intended. Also, we don't have the dataset that would be necessary for this purpose. Therefore, we decided to define a fixed structure for the command words and command forms. After having them fixed, we realized that it looks similar to a programming language code since in programming language code, every word is fixed. So, in the parsing step, we decided to use Lex and Yacc to parse the given string and detect the type of the command and its parameters.

In the Lex and Yacc stage, Clerk is converting the text coming from Speech-To-Text (STT) to a JSON object with the fields of command type and its corresponding parameters. Then, Clerk passes the generated JSON object to the client-side to execute the command.

Text To Speech (TTS)

The Text to Speech module is used to convert text messages coming from the client into the speech that is then written to an audio file. The audio file is played on the client-side. It also deletes the audio file back after the client-side plays it.

Copy and Paste

This module exists because the Word API doesn't provide a way to copy or paste text from inside the document. Instead, we send an indicator message from the client to the server that the user wants to use copy or paste operations, and when we receive the message we execute the operation accordingly.

4 Development/Implementation Details

Recorder

The recorder module we used is `node-record-lpcm16` [4]. This module was the one used in `Google Cloud Text to Speech API`'s tutorials since it was compatible with the Speech to Text API and also provided with an ISC license [5]. It has the following parameters: The sample rate at which audio is recorded, in Hertz; the silence threshold; the program to use for recording; the seconds of silence before ending; recording device; and the audio type to record. The important part is matching the sampling rate parameters for the recorder and the Speech to Text API client. The recording program used is `SoX` which is also the one used in tutorials and is open-sourced.

Lex and Yacc

Since we are working with Node.js in server side and TypeScript in client side, to implement such a Lex and Yacc, we needed a Lex and Yacc library in either Node.js or TypeScript. Therefore, we used *jison* library, which allows us to write Lex and Yacc in Node.js. With the help of *jison* library [6], "select from first word to third word" is converted to the format below.

```
{
    "select": {
        "SELECT": "select",
        "from_clause": {
            "index_word": {"index": 0, "WORD": "word"}
        },
        "to_clause": {
            "index_word": {"index": 2, "WORD": "word"}
        }
    }
}
```

It understands that it is a select command, it has a *from_ clause* and *to_ clause* and it converts "first" to "0", "third" to "2". It can be claimed that Lex and Yacc eased the implementation of the model classes. Therefore, Lex and Yacc plays a big role in the project architecture.

Lex

Lex is the pre-stage of Yacc and it tokenizes the words of the input string and sends the modified string to Yacc to be parsed. Therefore, Lex tokenizes all the reserved words in our application and it labels all the non-reserved words as *ANY_ WORD*. Below is given a several words and their corresponding tokens to represent the Lex structure.

```
    select              {return 'SELECT';}
    define              {return 'DEFINE';}
    find                {return 'FIND'}
```

```
next                {return 'NEXT';}
previous            {return 'PREVIOUS';}
current             {return 'CURRENT';}
[1-9][0-9]*         {return 'NUMBER';}
[a-zA-Z]+           {return 'ANY_WORD';}
```

Yacc

In Yacc step, the BNF starts from *start* branch, which lists all command type headings such as *select* and *delete*. These headings are branched in detail later on. Below is a list of several headings under *start* pointer to represent the structure of the BNF.

```
start:  select | delete | goto | read |
        repeat | copy | paste | save
```

Each of the command headings is extended to cover their phrases and an example shall be provided as follows.

```
select:
    SELECT |
    SELECT index_clause |
    SELECT FROM index_clause TO index_clause |
    SELECT ALL
```

In this example, we are listing the possible types of *select* command. To give an example, this structure allows the user to give "select first word of second sentence" command, where "first word of second sentence" are *index_ clause*s. Lex and Yacc labels it as "SELECT *index_ clause*". The definition of *index_ clause* branch is as follows.

```
index_clause:
    index_paragraph |
    index_sentence |
    index_word |
    index_word OF index_sentence |
    index_word OF index_paragraph |
    index_sentence OF index_paragraph |
    index_word OF index_sentence OF index_paragraph
```

Some examples for *index_ clause* can be given as "second word", "second sentence of third paragraph" and "first word of third sentence of last paragraph".

We have written the BNF structure such that it covers all possible command phrases in the same way of *select* command example. The full BNF for the application can be found in the Appendix section.

Speech to Text

Google's Cloud Speech-to-Text API is used in the Clerk for better accuracy and sustainability purposes. The API is compiling on the server-side of the project and after the server got

proper signals from the client-side via the socket, a `speechClient` object is created. The API is returning a stream after a request is done with proper credential values. The stream consists of possible outputs with their probabilities for a given speech. We are taking the most probable output from the API stream and our efforts to improve recognition accuracy are based on increasing that highest probability. To increase the recognition accuracy of command phrases, a speechContextPhrases variable is set in the configuration of `speechClient`. It consists of the commands phrases. If the probability of a command phrase is not the highest in the stream output, this variable increases that command phrase probability in the output file. For instance, "make bold" command generally recognized as "make old" by API, by adding "make bold" into the speechContextPhrases, we increase the probability of "make bold". Thus, "make old" probability goes down and API returns the wanted phrase. The interaction type, possible microphone distance, recording device type, and media type is also given to the configuration to increase the recognition accuracy accordingly.

Google Cloud Speech to Text API has a time limit of 5 minutes for any single connection. After 5 minutes, the connection is destroyed from the API side. To work around this issue, Google provides code snippets that help restart the connection after the timer is close to termination. They call this "infinite streaming". We utilized their code after modifying it to fit our use case.

The way infinite streaming works is that when a streaming limit is hit, the server destroys and restarts the `speechClient`. It takes time for the `speechClient`to restart in which the user may be still talking. To deal with this we have to store the sound in a buffer until the stream is available again. On top of that, we need to check whether the last packet of sound that was sent to the API was the final result of a transcription. If not, we need to re-send the last packet of sound together with the contents of the buffer.

Text to Speech

Google's Text-to-Speech API is used in the Clerk to interact with users and give them voice feedback. The API is compiling on the server-side of the project and after getting proper signals from Client-side via socket, a `textToSpeechClient` object is created and credentials are given. Server-side starts to wait until a Text-to-Speech request signal comes from the socket. Server-side send a request to the textToSpeechClient and get a binary audio content. This content is written to an mp3 file in the server directory. Then a signal is sent via socket to TextToSpeechLib class for playing the output mp3 file and after the play is done mp3 file is deleted. `Delete, Copy, Goto, Read` commands can create and send a signal for text-to-speech requests. They are creating this signal when a logic error occurs to inform the user that something went wrong and we could not complete your request. For example, a user may want Clerk to read from the 3rd sentence to the 6th sentence but there may be only 2 sentences in the file. In this situation Clerks warns the user that a logic error occurred and asks the user to reconsider its request.

Networking

Clerk uses `socket.io` node module for communication between the local server and client, and also to start the server[7]. This module allows easy to handle networking between the

local server and the client, which is the add-in application.

Copy and Paste

To execute copy and paste operations, Clerk uses `clipboardy` node module [8]. These operations are done on the server-side since the embedded browser inside Word doesn't allow access to the clipboard.

Commands

The classes in the program will be grouped according to their similar structure as follows.

RangeSelector Based Commands

RangeSelector class is one of the crucial classes in the program since its functionality provides a base for execution of other commands such as SpellCommand, CopyCommand, SelectCommand, ReadCommand, GoToCommand, and DeleteCommand. Since these classes use RangeSelector class methods extensively in their `execute()` methods, we have tried to implement many different types of helper functions to provide a very flexible concept so that it can allow finding a variety of different ranges the user needs to execute a command. Some of these commands are as follows: getting a paragraph at a given index, i.e third paragraph, getting next and previous words, sentences, and paragraph and much more (See 3.2.5). Since these classes have almost the same structure to execute to the method, we will try to explain different implementation details with only one of them.

During the implementation of the RangeSelector class, we benefit from the Word JavaScript API that Microsoft has provided for developers. It does allow us to find the range of word, sentence, and paragraph where the cursor is. That is a downside of the API. We will explain it in more detail later (See 4.7). Assume that we want to find the second sentence of the current paragraph. There is no ready function to do that so most of the time to find ranges we did extra process such as splitting the paragraph into sentences. This same idea applies to find words, and paragraphs in the document. We loop over the current paragraph to find the second sentence. Now, suppose that the user has a complex command like `"delete the second word of the last sentence in the previous paragraph"`. We treat `delete` part that is the command type as a generic part and process the rest of it as an index clause. We extract indices for word, sentence, and paragraph.

A command could be something like this: `"copy from the last sentence of the previous paragraph to the first word of the fourth sentence of the next paragraph"`. `parseTree` that comes with the constructor contains the necessary index clauses which are done by the parser as well as the command type. If we return to our previous example, we extract these index clauses from our `parseTree`. So, now we need to find the range for the from-to type of index clause and we have implemented another method `getRangeFromTo()`. Since we know the beginning and end of the wanted range, starting from the beginning we

expand our range until it reaches the end.

Another case for the command is just saying the name of the command like `spell, read, copy, delete` without any index clause to process. In these cases, the selection of the range has been already done manually by the user so we use appropriate methods that have been implemented in the RangeSelector class. One exception is that for SpellCommand if there is no selection done by the user, the program sends the current word where the cursor is to TextToSpeechLib. Another thing is that we know that at the very beginning we only get a `parseTree`, yet, even though we know the possible command structure and index clauses, we do not know which exactly. This is because we have tried to generate our `parseTree` as generic as possible so that we ease the process of command execution in every class. Therefore, the first thing we do is that we unpack the `parseTree` and use the if-else structure to handle the cases and do what is needed in each part because they might and will require different operations. For example, the command `"go to the beginning of the second sentence"` moves the cursor into the wanted position, so other commands do their related jobs. Also, for some reason, a command cannot be executed a warning will be read to the user and printed to the error log.

Browser Based Commands

The following classes have a very similar structure since they require web search in their implementation: GoogleCommand, TheasurusCommand, and DefineCommand. When any of these commands are executed, the program will detect the sub-type of the `parseTree`, which is sent by the Parser class (See 3.1 for more detail). Assume the sub-type is detected as `google` and the user has already selected a piece of text manually, we redirect this request to google. If there is no selected text, we take the current word to be searched on google. Furthermore, if the selected text or current word is already a website URL, we open that website in the browser, otherwise, the program searches that text in Google.

For DefineCommand, instead of searching on Google we look up in Cambridge Dictionary for the selected text or current word depending on whether the user has already selected a text or not. For TheasurusCommand, we search the selected text or current word in Thesaurus for finding similar words.

Mode Based Commands

The program needs to have different states with different permissions and capabilities. If the program writes everything we said to the document, that would be an unwanted situation, to prevent similar problems, the program needs clear distinctions between the states in terms of what and when to do: "There is a time and place for everything". As we have mentioned earlier, we have three different states: `SLEEP`, `SPEECH`, and `COMMAND`. The following classes are responsible for setting the correct state of the program when needed: CommandMode, StopListening, SpeechMode, SwitchMode, and WakeUp. These classes act as nothing but a setter method. CommandMode changes the state of the program from `SPEECH` to `COMMAND` so that the user will be able to execute commands afterward. `StopListening` changes the state

of the program from `COMMAND` to `SLEEP` so that the user will be able to execute commands afterward. SpeechMode changes the state of the program from `COMMAND` to `SPEECH` so that the user will be able to insert text into the document afterward. SwitchMode is responsible for switching the state of the program between `SPEECH` and `COMMAND` as a response to the keypress that is determined. WakeUp changes the state of the program from `SLEEP` to `SPEECH` so that the user will be able to insert text into the document afterward.

Other Commands

When FontCommand is executed, Clerk first checks if there is a selected range. If there is, it changes the font values of the selected range. Otherwise, it stores the values in the corresponding static variable and when an InsertCommand wants to insert a text, Clerk changes the text's font values accordingly and then inserts the text to the document. Due to this architecture, our FontCommand is working slightly different than font buttons of the Word application.

In the execution method of InsertCommand, Clerk first detects the sub-type of the parsed tree which is sent by the Controller class. Sub-type can be `insert new line, insert new paragraph, insert new page, or insert any`. When the detected sub-type of the parsed tree is `insert new line`, Clerk adds a new line into the document. If the sub-type is `insert new paragraph`, Clerk adds a paragraph and if the sub-type is `insert new page`, it adds a page into the document. If the sub-type is not detected as one of them, Clerk inserts the text directly into the document. Assume the program received a command like `insert hello world`. Firstly, the type of the command, which is `insert` in this case, will be detected by Parser Class, then the parsed tree will be sent to the InsertCommand. After InsertCommand receives the parsed tree, InsertCommand will detect the sub-type of the parsed tree. Since the sub-type is not one of `new line, new paragraph, or new page`, Clerk will insert `hello world` into the document directly.

When SaveCommand is executed, we are sure that the sub-type of the command is `save` since there is only one sub-type in this class. After execution, Clerk will save everything written in the document.

In RepeatCommand class, Clerk first detects the sub-type of the parsed tree which is sent by Parser class. It can be `repeat` or `repeat times`. Assume the sub-type is detected as `repeat`, Clerk will repeat the last command executed successfully one more time. If the sub-type is detected as `repeat times`, Clerk will repeat the last command several times that is indicated in the parsed tree. For example, we have a command like `repeat five times`, Clerk will repeat the last executed command five times.

When FindCommand is executed, the sub-type of FindCommand is `find` for sure since there is only one sub-type in the class. After detection, Clerk will find the appearances of the keyword in the text field of the parsed tree throughout the document. Assume we have a command like `find hello`, the program will find all of `hello` words in the document and

highlight the first `hello` word of the document. The user may want to find other appearances of the word too, for this case we have functions that find the next or previous appearances of the word. `find next` command will highlight the next one and `find previous` command will highlight the previous one. In this example, `find next` will highlight the next `hello` word and `find previous` will highlight the previous `hello` word.

Word API

One of the components on which the program depends is Word JavaScript API. Therefore, its capabilities affected heavily our implementation of the program. So, if there is a functionality that we need to implement and does not exist in the API, we had to find a way to accomplish what we have got. For this, an example could be the copy and paste commands. After selecting the appropriate text, we have forwarded it to the server where it will be copied to the clipboard. When paste command is requested, on the server side we get the text from the clipboard and sent it to the client-side and insert it into the document. The same idea applies to read command as well because the API does not provide TextToSpeech functionality even though they have it in the MS Office 365. Another example would be moving the cursor in different places. Assume that we have a command `go to the second sentence of the first paragraph`. Now, Word API does not provide a function to do this directly. Hence, we are splitting the words, sentences, and paragraphs to find the appropriate position. Afterward, we can move the cursor there and execute a variety of commands. Also, the API used synchronization in the functions extensively, therefore, we also had to implement our methods accordingly. Numerous examples can be seen in the code, one is that `await` keyword that states that the program sequence should wait for the completion of that line to go forward.

Besides, there are some functionalities that we wanted to implement but could not do it because of Office API, general API of Microsoft Office applications. One example would be to Undo and Redo commands. We found out that Microsoft did not implement this functionality in Office API because some operations cannot be undone in Excel. Therefore, they did not want to allow the add-ins of the Microsoft Office application to reach the undo stack. Due to this problem, although we have thought of implementing `Undo` and `Redo` commands, we were not able to implement them. Another example can be given as page and section navigation commands. The API was not allowing us to get the page and section information of a range. Therefore, it was not possible to implement `"go to third page"` command or `"go to next section"`. Although we know that it would be so useful for the user, we couldn't supply this functionality.

5 Testing Details

Microsoft Word application runs the add-in on the Edge browser. To debug an Edge browser page, we needed the console to see our logs. However, Microsoft Word does not provide a console window for add-in developers. They provide another application for add-in developers, called Microsoft Edge Devtools Preview[9]. This tool provides a console, debugger, and other tools necessary for the development of applications that run on the Edge browser. We

utilized this tool during development and testing.

We tested the speech to text component by isolating the module from the server, along with the recorder. Google provides many different features for the Cloud Speech to Text API that can make it more specific to different use cases like speech adaptation and boost. We tested the Speech to Text module with many different settings to obtain the configuration that we felt was best for our application.

After the development of the add-in we tested it as users to see how well it functions since it is a speech-based application.

6 Maintenance Plan and Details

As Clerk and its features get exploited by users more and more, new ideas and enhancements will reach our feedback channels, and also bugs will be reported by our users. The project will use Bugzilla as issue tracker after the first release since its an open-source tool, it is still under active development, it allows plenty of other tools to integrate, and most importantly it is free. We decided to use a corrective maintenance approach. When a user complains about a feature of Clerk and reports that it is not working as expected. Our testing team will try to reproduce it and if the user is right about the feature is not working as expected, a bug will be created in the Bugzilla system. The whole team will be informed via communication channels like Telegram which will be integrated into the Bugzilla system. The priority and severity of the bug will be set and a developer or developers will be assigned to the bug according to bug's priority and severity and workload of the team members. The point of this corrective maintenance approach is to perform changes to the software to keep it with its original, planned requirements and specifications.

7 Other Project Elements

7.1 Consideration of Various Factors

Public Health Considerations

Provided that it works efficiently and accurately enough, Clerk will have a positive impact on public health. When people are using a program such as Word and typing consistently for a long period, they tend to sit in ways that are bad for their posture. Adding to that, they also need to sit in front of a screen and directly look at it for a long period. But with this type of add-in users can have the freedom of sitting or standing however they want while they are working on editing a document and they don't have to look at the screen to the same degree. If more of the same type of programs are built for different platforms, then the impact will be even greater.

Additionally, for people with visual impairment problems or other disabilities who can't use keyboard and mouse effectively, Clerk and similar applications can make their day-to-day activities easier by removing some restrictions from their lives. It is a well-known fact that empowering people with disabilities results in happier individuals and happier people are generally healthier.

Public Safety Considerations

Our project does not have any clear contributions to public safety; nor does it pose any threat to it.

Public Welfare Considerations

Clerk can improve the welfare of society by making its users healthier and happier as mentioned in the public health considerations. Empowering disabled people will also improve our welfare. Human beings have an inner motive to work and create things. Enabling more people to participate in different tasks such as writing reports or creative tasks like writing books will improve society as a whole.

Global Scale Considerations

We have considered that more people can use our add-in on a global scale if we provide it mainly in the English language, rather than our native tongue of Turkish, as we have five Turkish members. Therefore, at this point we are going to build it in a way that works with English. However, speech recognition API's and libraries are advanced enough to recognize more languages than just English. Though we want to prioritize making the application work with the English language first, it could be a consideration later to make it multilingual in the future.

Environmental Considerations

Our project doesn't have any clear, significant effects on the environment, negative or positive.

Economic Considerations

We have decided that we will provide our add-in to Word users free of charge. However, it should be considered that it already costs a certain amount of money to obtain an Office365 subscription which is required to use Word with all its functionality. At the same time, our add-in will also be available on Office Online which can be used free of charge, though it doesn't quite provide all the functionality that subscribers get on a desktop version.

Our decision to make the add-in free hinges on the fact that right now we are considering a design that won't cost us any money to maintain. This design decision may change in the future which would mean the economic considerations may be revisited.

Cultural Considerations

We considered that a very popular application like Word almost comes with its own culture and user behavior. We considered what a typical Word user would want a feature to work like with voice inputs. We thought about the keywords to use and the convenience of them. Since our application's goals are aligned with improving the user's quality of life while using Word, these are very important considerations.

Social Considerations

Our considerations in terms of the social impact of our project and the effect of society on it have already been mentioned in public health, welfare, global, economic, and cultural considerations sections.

	Effect Level	Effect
Public Health	6	More relaxed ways of using a text editor and empowerment of people with disabilities.
Public Safety	0	None
Public Welfare	5	Empowerment of people with disabilities.
Global Scale	6	Use of English language.
Environmental	0	None
Economic	4	Providing the add-in free of charge.
Cultural	6	Thinking about how users would use Word with voice.
Social	8	Combination of health, welfare, global scale, economic and cultural effects.

Table 1: Factors that have affected the design and their severity.

7.2 Ethics and Professional Responsibilities

Law number 6698 on the Protection of Personal Data known as KKVK in Turkey is holding us responsible to keep personal information private and not to share this personal information with any third-party. Since, we will not hold any personal data, we will not have any problem related to this.

One of the very first responsibilities is to accept individual responsibility. Even though work-packages mentioned above have leaders, following orders does not justify approving bad designs for other members. Having a leader in a work-package does not necessarily mean that he does what he wants. One cannot always be a team player. At this point professional standards have priority. If there is something to argue, we will make it the subject of a meeting to make the product better.

As software engineers to be, we are responsible for looking beyond the customer's opinions. We should have a precise description of any problem that may occur in the project. We should get that problem reviewed by other members before building to solve the real problem.

Another professional responsibility that we have to meet is that we should be honest about the capabilities of our project. We will not offer technical solutions where there are none. And if there is a solution to a specific problem at this moment, we will talk about the possibility of a solution, avoid giving precise statements about a solution.

Another important responsibility for us is to produce review-able designs for our projects. This will allows us to keep good documentation throughout the project to create a communication channel among the members and producers and customers.

7.3 Judgements and Impacts to Various Contexts

We decided to use English as the recognition language based on the global scale considerations posted in section Global Scale Considerations. This should have an impact on a global scale. This is described in the table below.

Judgement Description	Decision to use English.	
	Impact Level	Impact Description
Impact in Global Context	10	English is the most used language around the globe, so it makes sense to use it.
Impact in Economic Context	0	None.
Impact in Environmental Context	0	None.
Impact in Societal Context	0	None.

Table 2: Describing the global, economic, environmental and societal impacts of choosing English as the main language of use for the program.

We decided that Clerk would be an open-source and free to use add-in application based on the discussion in section Economic Considerations. This should have a mainly economic impact but also societal too in terms of availability.

Judgement Description	Making the add-in open source and free to use.	
	Impact Level	Impact Description
Impact in Global Context	0	None.
Impact in Economic Context	10	The add-in costs no money to use.
Impact in Environmental Context	0	None.
Impact in Societal Context	6	The add-in will have more availability.

Table 3: Describing the global, economic, environmental and societal impacts of making the add-in open source and free to use.

We determined that our application didn't need to have any user information stored. This should have some societal and global impact in terms of the privacy of users.

Judgement Description	Making the add-in open source and free to use.	
	Impact Level	Impact Description
Impact in Global Context	8	No privacy issues.
Impact in Economic Context	0	None.
Impact in Environmental Context	0	None.
Impact in Societal Context	8	No privacy issues.

Table 4: Describing the global, economic, environmental and societal impacts of storing no user information.

7.4 Teamwork and Peer Contribution

Teamwork

In the requirements and design stages, we met every few weeks to discuss details of the design and functionalities of the application, and which technologies we could use in the implementation stage to make our design a reality.

In the implementation stage, our division of work was based on different parts that we each worked on. In this way, we could work more efficiently than everyone working by themselves while also avoiding meeting place and time problems that would be raised by all of us working together at once. We tried to do pair programming as much as possible, though this proved difficult when we were forced to work remotely after the university was suspended due to the COVID-19 outbreak. After starting to work remotely, we still kept up our meetings but the efficiency of our work, unfortunately, but unavoidably, took a hit.

Throughout the semester we tried our best to hold weekly or bi-weekly meetings with all members who could attend at that time.

Peer Contribution

Samet Demir:

- implemented Lex and Yacc
- implemented relation between `Client`, `Controller` and `Parser` and `Command` classes
- helped implementation of Text-to-Speech
- helped implementation of `execute()` methods of the most of the commands
- implemented `RangeSelector` class
- helped implementation of `Parser`
- helped implementation of User Interface

Ahmet Malal:

- helped implementation of `RangeSelector` class
- implemented the `execute()` methods of Copy, Delete, Google, GoTo, Insert, Paste, and Select commands.
- studied Word API to implement the command classes above.
- tested the implementation of command classes.

Ensar Kaya:

- helped implementation and integration of Text-to-Speech API
- helped implementation and integration of Speech-to-Text API

- helped implementation of User Interface
- helped implementation of Socket framework

Faruk Şimşekli:

- helped implementation of `RangeSelector` class
- implemented the `execute()` methods of Copy, Delete, Google, GoTo, Insert, Paste, and Select commands.
- studied Word API to implement the command classes above.
- tested the implementation of command classes.

Muhammed Salih Altun:

- Implemented server-side speech recognition and helped improve it.
- Studied webpack to understand how it works in building the application.
- Helped implement parsing logic on server and client-side.
- Helped implement Thesaurus, Spell, Define commands.
- Helped with testing and finding bugs.
- Implemented infinite streaming on the server side.

7.5 Project Plan Observed and Objectives Met

Earlier time of the year while we were planning the project, after risk management, we tried to distribute work packages throughout the year. We were aware that proper control of the schedule requires the careful identification of tasks to be performed and accurate estimations of their duration, the sequence in which they are going to be done, and how people and other resources are to be allocated. Any schedule should take into account vacations and holidays.

Despite of careful planning, some problems have occurred, one being speech to text permission related. We were not able to convert and type it into the document. Up to this point, work package 2, we had some design in our project. After an extensive search, consultation with an engineer who had the same problem, we found out that this is unsolvable unless you ask Microsoft. So we had to come up with another way to accomplish it, which took some time. This problem delayed our plan for one and a half months. So, we were behind the schedule most of the time. But, the problems kept coming. A risk that no one could have foreseen: COVID-19. That hit hard our plan.

Project management is not an easy job. The motivation of one member affects the project. We knew that motivation helps people work more efficiently and produce better results, however, with the virus problem, we have lacked a serious amount of motivation and we could not put some work into the project at least the first three weeks. After this point, we were already more than two months behind the schedule and needed to move. We tried to work as much as possible not to miss the deadline. In the end, we caught it and we have met our objectives with a few casualties that are not enabling to implement a couple of functionalities.

7.6 New Knowledge Acquired and Learning Strategies Used

Before doing this project, we did not have so much knowledge about JavaScript, HTML, and CSS. We learned what is the relation between them by searching on the Internet.

We were not familiar with the Word API and we have read almost all of the documentation of Microsoft about Word API to implement our functionalities and to improve our program.

We have learned that it is possible to implement Lex and Yacc in JavaScript with the help of *jison* library [6]. And we learned the structure and syntax of the library by reading their documentation.

We have learned the Server-Client architecture and since we have server and client programs running at the time, we learned how to send data from one program to another.

We were familiar with design patterns from the OOP course, and we put our knowledge into the product to write a cleaner and modular code. We We learned how to use *webpack* module bundler to build applications. As a learning strategy, we used tutorials from webpack's website [10].

We learned how to use Google Cloud Text to Speech and Speech to Text APIs, along with a recorder. To learn, we utilized tutorials from Google Cloud's website [11]. We also learned how to use the cloud console to activate APIs and use the generated API keys.

We improved our knowledge of version control systems and Git with Github workflow through gaining more experience and working together and also using tutorials. Research was a big part of the project to learn new things. Other than that, most of the time we used hands-on experience to learn a new subject.

8 Conclusion and Future Work

In conclusion, we designed and built a Word add-in that functions as somewhat of a text editor assistant that can work with speech commands. We have implemented the commands that we set out to implement, so long as the Word API allowed it. Our add-in, Clerk, can make the lives of a lot of users easier.

For the future of our project, we can add more commands that are available in the Word API. When Word API allows the developers to have more functionalities, we can add Redo and Undo, section navigation and page navigation commands. If the range of functionalities that Word API gives access to increases, then we can implement those as well. We can also try to use different speech models and APIs to increase speech recognition accuracy or response time. Furthermore, we could use Neural Networks that improves with constant use by one user and adapt to understanding them better. Also, we could use Natural Language Processing (NLP) for parsing to make Clerk more flexible with the command words. We can also deploy our add-in to the Microsoft Store.

9 Glossary

Word add-in: An application that can be built and embedded inside Microsoft Word and work inside a Word instance that can extend the available functionality.

Node.js module: Similar to a software library, a package that includes functions that can be executed in the Node environment.
Speech to Text: Services that enable conversion of sound segments into the textual transcription of the words spoken in them.
Text to Speech: Services that convert text to spoken language audio.
Lex: A program that generates lexical analyzers based on some rules.
Yacc: A commonly used parser generator.
Context-Free Grammar: A set of recursive rules used to generate patterns of strings.
MVC: Model - View - Controller architecture
Jison: A library to write Context-Free Grammar in JavaScript
JSON: JavaScript Object Notation. A special representation of objects in JavaScript that is used to store objects.

10 Appendix

10.1 User Manual

In this appendix, a user manual is presented for a better understanding of Clerk add-in.

Default Home Page

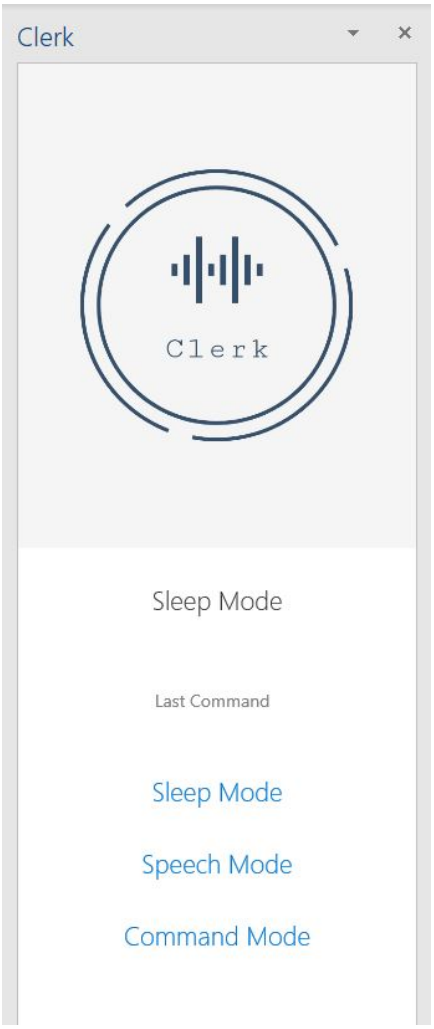

Figure 7: The Home Page at the beginning

The user interface of the Clerk consists of 1 icon at the top, 1 row for show user the current mode of the add-in, 1 row for showing what the add-in heard so far, in other words, interim results, 1 row for showing recognized command, and 3 buttons for switching between modes. The sleep mode listens to the user but not typing anything on the document, it waits for the user to say "wake up" or change the mode to the speech or command. Speech mode listens to the user and types everything it hears into the document including punctuation marks. The command mode is unique to Clerk and it consists of plenty of Word functionalities that are going to be explained in detail in further pages.

Home Page-Speech Mode

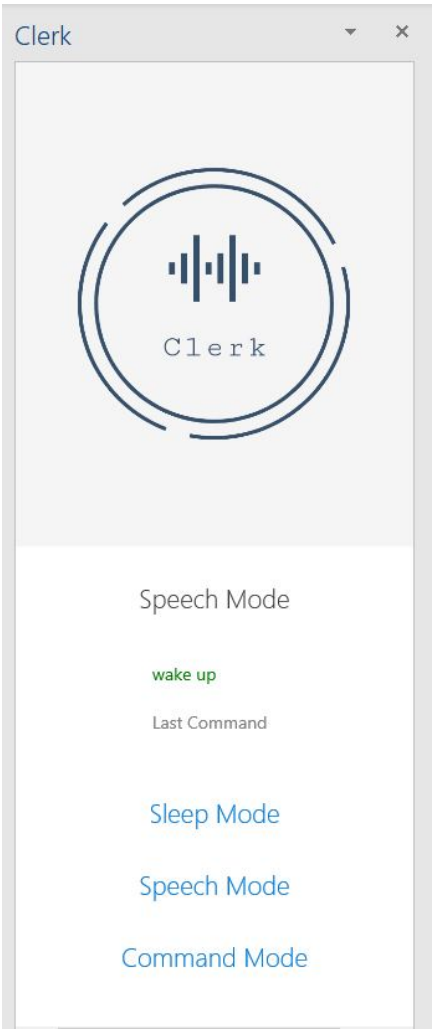

Figure 8: The Home Page when wake up command recognized

In sleep mode, the Clerk listens and waits for user to say "wake up" or click speech mode buttons. Green color means that it is the final result.

After switching to the speech mode, Clerks start to type what it hear into document. For example, the figure 9 shows recognized "hello world" words.

But that row might be red sometimes for example when user is still speaking, Clerk shows the interim results with red color. When they become final, the color turns to the green.

The space bar can be used instead of buttons to switch between command and speech modes.

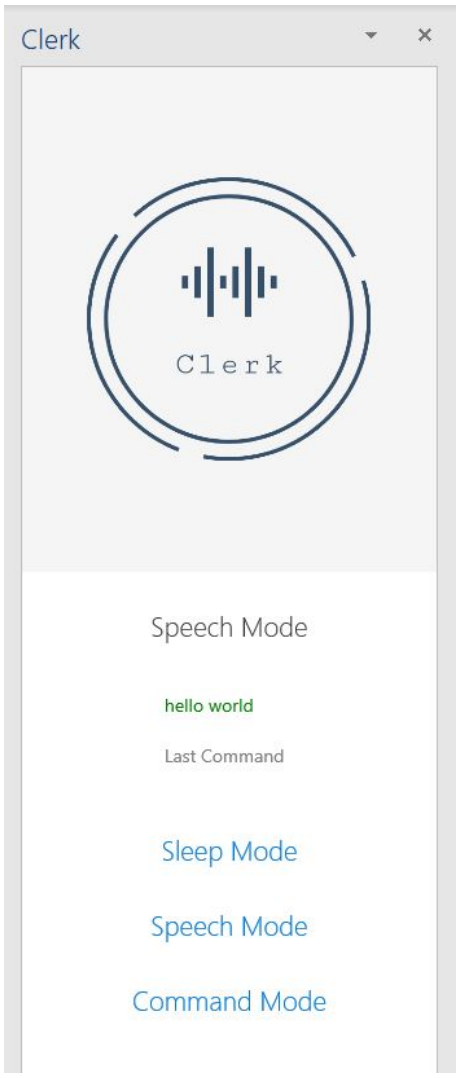

Figure 9: The Home Page when "hello world" recognized

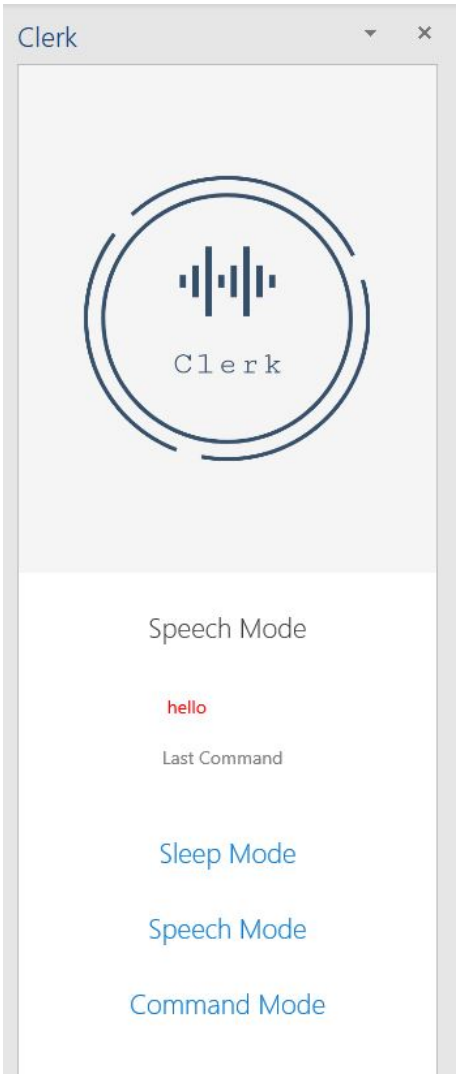

Figure 10: The Home Page when "hello world" is not recognized yet

Home Page-Command Mode

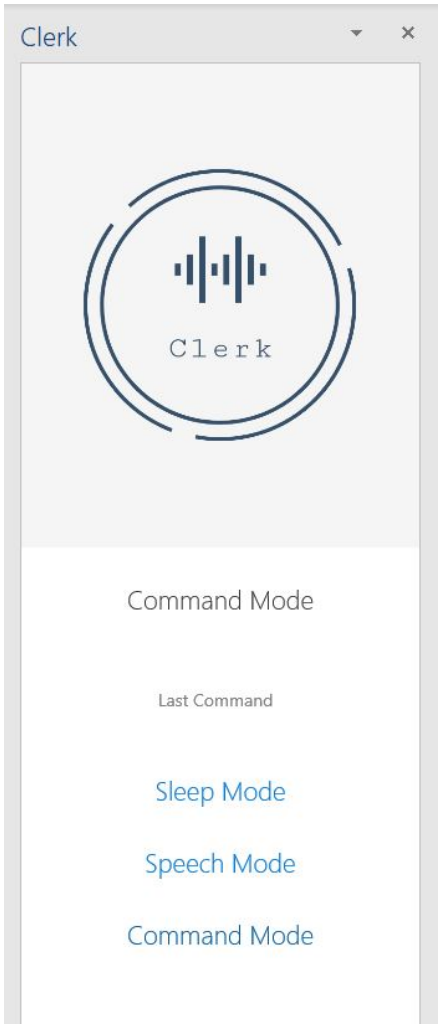

Figure 11: The Default Home Page in Command Mode

The default page for command mode has no difference from other mode home pages. The command mode is designed for people who are not able or do not want to interact with keyboard with the purpose of increasing the usability of the MS Office Word's functionalities. The optimal goal is to allow users to easily use all necessary functionalities of Word with high accuracy and without any unexpected behaviours.

When a command is not recognized, the Clerk shows the command with red color. When a command is recognized, the Clerk shows the result with green color and updates last command.

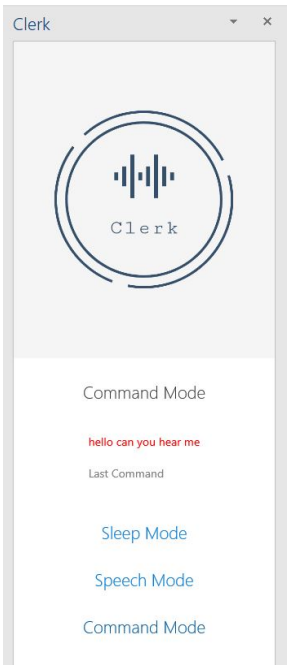

Figure 12: The Home Page when a command is not recognized

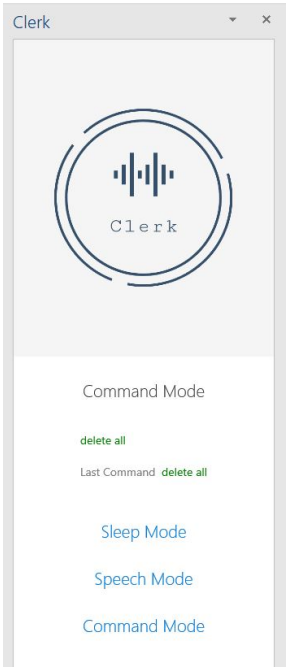

Figure 13: The Home Page when a command is recognized

Command Mode: Possible Commands and Their Usage

Definitions:

- *index* : It has the following keywords to describe index of word, sentence, or paragraph. For simplicity, *index* can be at most ten. It can be extended to hundreds as a future work.
  - First/Second/Third/Fourth/Fifth
  - Sixth/Seventh/Eight/Ninth/Tenth
  - Last
- *index_ clause* : It can take index values or navigation values and the word, sentence, or paragraph words. E.g.
  - First/Second word
  - Next/Previous sentence
  - Second sentence of last paragraph
  - Fifth word of the fifth sentence
  - Third word of first sentence of first paragraph
  - Third word of first sentence of next paragraph
- *from_ clause* : It starts with "FROM" keyword which is followed by a *index_ clause*.

  - FROM First/Second word
  - FROM Next/Previous sentence
  - FROM Second sentence of last paragraph
  - FROM Fifth word of the fifth sentence
  - FROM Third word of first sentence of first paragraph
- *to_ clause* : It start with "TO" keyword which is followed by a *index_ clause*.
  - TO First/Second word
  - TO Next/Previous sentence
  - TO Second sentence of last paragraph
  - TO Fifth word of the fifth sentence
  - TO Third word of first sentence of first paragraph
- *navigation* : It has following keywords for navigation.
  - Next
  - Previous
  - Current/This
- *font_ word* : It has following keywords which describes the font specific keywords.
  - Bold/not Bold
  - Italic/ not Italic
  - Underline/ Underline none
  - Red/Blue/Yellow/Black/Green
  - Size 10/ Size 12/ Size 14
- *font_ words* : It is a list of *font_ word* phrases
  - not Bold Underline Red
  - Underline none Blue Size 15
  - not Bold Italic Green Size 25 Underline
- *times* : It can take values for repetition purposes. This value can be at most 10.
  - Once/Twice/Three times/Fourth times/Fifth times
  - Sixth times/Seventh times/Eight times/Ninth times/Tenth times

Copy: There are 6 possible ways to use this command.

- copy → The command copies the selected part into the clipboard.
- copy all → The command selects all document and copies into the clipboard.

- copy word → The command copies the word where the cursor is into the clipboard.
- copy sentence → The command copies the sentence where the cursor is into the clipboard.
- copy paragraph → The command copies the paragraph where the cursor is into the clipboard.
- copy *index_ clause* → The command copies the range given by *index_ clause* into the clipboard.

Define: This command searches words selected by the user or the word where the cursor is in the Cambridge Dictionary web page according to selection situation. If selection is empty it searches the current word. "Define" is the special word to execute this command.

Delete: There are 7 possible ways to use this command.

- delete → The command deletes the selected part from the document.
- delete all → The command selects all document and deletes from the document.
- delete word → The command deletes the word where the cursor is from the document.
- delete sentence → The command deletes the sentence where the cursor is from the document.
- delete paragraph → The command deletes the paragraph where the cursor is from the document.
- delete *index_ clause* → The command deletes the range given by *index_ clause* from the document.
- delete *from_ clause to_ clause* → The command deletes the range between *from_ clause* and *to_ clause* from the document.

Find: This command searches the word/words selected by the user or the word where the cursor is in the whole document. It is also able to iterate among the founded words.

- find hello world → The command finds and marks the first appearances of "hello world" in the document.
- find [next | previous] → The command iterates to the next/previous appearance of the searched phrase in the document.

Font: This command sets the font properties of the document. If there is a selected text, the changes are only effects selected parts. If nothing was selected, the general font setting will be adjusted. There are 3 possible ways to use this command.

- *font_ words*
- Set font *font_ words*

- Make *font_ words*

Google: This command searches words selected by the user or the word where the cursor is in the Google web page according to selection situation. If selection is empty it searches the current word. If selection is a web page URL, it opens that page. "Google" is the special word to execute this command.

GoTo: There are 3 possible ways to use this command. It basically moves cursor to the desired location.

- Go to *index_ clause* → The command moves the cursor to the end of the *index_ clause* location.
- Go to [beginning | end] of *index_ clause* → The command moves cursor to the beginning/end of the *index_ clause* location.
- Go to [beginning | end] → The command moves cursor to the beginning/end of the selected text or beginning/end of document if the selection is empty.

Insert: There are 3 possible ways to use this command. It adds white space characters into the document.

- Insert new line → The command adds a new line and moves the cursor to that location.
- Insert new paragraph → The command adds a new line and a tab, finally moves the cursor to that location.
- Insert new page → The command adds a new page and moves the cursor to that location.

Paste: There are 2 possible ways to use this command.

- Paste → The command paste the text in the clipboard to the document.
- Paste *times* → The command paste the text in the clipboard to the document several times described with the "times".

Read: There are 9 possible ways to use this command.

- Read → The command reads the selected part from the document.
- Read all → The command selects all document and reads from the document.
- Read word → The command reads the word where the cursor is from the document.
- Read sentence → The command reads the sentence where the cursor is from the document.
- Read paragraph → The command reads the paragraph where the cursor is from the document.

- Read *index_ clause* → The command reads the range given by *index_ clause* from the document.
- Read *from_ clause to_ clause* → The command reads the range between *from_ clause* and *to_ clause* from the document.
- Stop Reading / Stop → This command stops the current reading permanently.

Repeat: This command re-runs the last successfully recognized command several times.

- Repeat → The command re-runs the last command.
- Repeat *times* → The command re-runs the last command *times* value times. If *times* is "five times", it will repeat the command 5 times.

Save: This command saves the document as the Ctrl+S shortcut. "Save" and "Save file" are the special words to execute this command.

Select: There are 5 possible ways to use this command.

- Select all → The command selects all document.
- Select word → The command selects the word where the cursor is.
- Select sentence → The command selects the sentence where the cursor is.
- Select paragraph → The command selects the paragraph where the cursor is.
- Select *index_ clause* → The command selects the range given by *index_ clause*.
- Select *from_ clause to_ clause* → The command selects the range between *from_ clause* and *to_ clause*.

Spell: This command spells the words selected by the user or the word where the cursor is by using Text-to-Speech API. If selection is empty it spells the current word. "Spell" is the special word to execute this command.

Thesaurus: This command searches words selected by the user or the word where the cursor is in the Thesaurus web page according to selection situation. If selection is empty it searches the current word. "Thesaurus" is the special word to execute this command. Thesaurus finds the similar words to the selected text.

Stop Listening: This command switches command mode to the sleep mode. "Stop Listening" is the special word to execute this command.

10.2 Full Context-Free Grammar for Command Parsing

```
start: command

command: select | delete | goto | read | insert | find | font |
         repeat | copy | paste | save | spell | define | thesaurus |
         google | mode

select: SELECT index_clause |
        SELECT from_clause to_clause |
        SELECT ALL |
        SELECT WORD |
        SELECT SENTENCE |
        SELECT PARAGRAPH

delete: DELETE index_clause |
        DELETE from_clause to_clause |
        DELETE |
        DELETE ALL |
        DELETE WORD |
        DELETE SENTENCE |
        DELETE PARAGRAPH

goto: index_clause |
      selectType OF index_clause |
      selectType |
      GOTO index_clause |
      GOTO selectType OF index_clause |
      GOTO selectType

read: READ index_clause |
      READ from_clause to_clause |
      READ |
      READ ALL |
      READ WORD |
      READ SENTENCE |
      READ PARAGRAPH |
      STOP READING |
      STOP

insert: NEWLINE |
        NEWPARAGRAPH |
        NEWPAGE |
        INSERT NEWLINE |
        INSERT NEWPARAGRAPH |
```

```
        INSERT NEWPAGE |
        INSERT any

find: FIND any |
      FIND

font: font_words |
      SETFONT font_words |
      MAKE font_words

repeat: REPEAT |
        REPEAT times |
        DOIT times |
        AGAIN

copy: COPY index_clause |
      COPY |
      COPY ALL |
      COPY WORD |
      COPY SENTENCE |
      COPY PARAGRAPH

paste: PASTE times |
       PASTE

save: SAVE

spell: SPELL

google: GOOGLE

define: DEFINE

thesaurus: THESAURUS

mode: listen | stop | switch | wakeup | commandMode

listen: START LISTENING | LISTEN

stop: STOP LISTENING | DONT LISTEN | DO NOT LISTEN

switch: SWITCH MODE | SWITCH

wakeup: WAKEUP | START DICTATION
```

```
commandMode: COMMAND MODE | COMMAND

from_clause: FROM index_clause

to_clause: TO index_clause

selectType: BEGINNING | END

index_clause: index_paragraph |
              index_sentence |
              index_word |
              navigation_paragraph |
              navigation_sentence |
              navigation_word |
              index_sentence OF index_paragraph |
              index_sentence OF navigation_paragraph |
              index_word OF index_sentence |
              index_word OF navigation_sentence |
              index_word OF index_paragraph |
              index_word OF navigation_paragraph |
              index_word OF index_sentence OF index_paragraph |
              index_word OF index_sentence OF navigation_paragraph |

index_word: index WORD

index_sentence: index SENTENCE

index_paragraph: index PARAGRAPH

navigation_paragraph: navigation PARAGRAPH

navigation_sentence: navigation SENTENCE

navigation_word: navigation WORD

font_words: font_word font_words |
            font_word

font_word: BOLD |
           ITALIC |
           UNDERLINE |
           NOT_BOLD |
           NOT_ITALIC |
           UNDERLINE_NONE |
           COLOR |
```

```
           SIZE_NUMBER

navigation: NEXT | PREVIOUS | CURRENT

index: LAST | FIRST | SECOND | THIRD | FOURTH | FIFTH |
       SIXTH | SEVENTH | EIGHTH | NINTH | TENTH

times: ONCE | TWICE | THREE_TIMES | FOUR_TIMES | FIVE_TIMES |
       SIX_TIMES | SEVEN_TIMES | EIGHT TIMES | NINE_TIMES | TEN_TIMES

any: special_words any |
     ANY_WORD any |
     ANY_WORD |
     special_words

special_words: SELECT | DOIT | FROM | TO | NEXT | PREVIOUS | CURRENT |
               WORD | SENTENCE | PARAGRAPH | ALL | LAST | FIRST |
               SECOND | THIRD | FOURTH | FIFTH | SIXTH | SEVENTH |
               EIGHTH | NINTH | TENTH | HERE | DELETE | THE | INSERT |
               BOLD | ITALIC | UNDERLINE | MAKE | NOT | DONT | SETFONT |
               COLOR | NONE | REPEAT | AGAIN | DO | REDO | UNDO | ONCE |
               TWICE | THREE_TIMES |FOUR_TIMES | FIVE_TIMES | SIX_TIMES |
               SEVEN_TIMES | EIGHT_TIMES | NINE_TIMES | TEN_TIMES |
               THIS | COPY |PASTE | READ | GOTO | NO | COMMAND | SPEECH |
               SLEEP | WAKEUP | SWITCH | START | DICTATION | STOP |
               LISTENING | LISTEN | NOT_BOLD | NOT_ITALIC | UNDERLINE_NONE |
               MODE | NUMBER | SAVE
```

References

[1] "Office Add-ins Documentation - Office Add-ins | Microsoft Docs," https://docs.microsoft.com/en-us/office/dev/add-ins/, Accessed: 2019-10-12.

[2] "Word JavaScript API Overview - Office Add-ins | Microsoft Docs," https://docs.microsoft.com/en-us/office/dev/add-ins/reference/overview/word-add-ins-reference-overview?view=word-js-preview, Accessed: 2019-10-11.

[3] "office package - Office Add-ins | Microsoft Docs," https://docs.microsoft.com/en-us/javascript/api/office?view=word-js-preview, Accessed: 2019-10-13.

[4] "node-record-lpcm16 - npm," https://www.npmjs.com/package/node-record-lpcm16, Accessed: 2020-05-26.

[5] "ISC License (ISC) | Open Source Initiative," https://opensource.org/licenses/ISC, Accessed: 2020-05-26.

[6] "Jison," https://github.com/zaach/jison, Accessed: 2020-2-18.

[7] "socket.io - npm," https://www.npmjs.com/package/socket.io, Accessed: 2020-05-26.

[8] "clipboardy-npm," https://www.npmjs.com/package/clipboardy, Accessed: 2020-05-27.

[9] "Get Microsoft Edge DevTools Preview - Microsoft Store," https://www.microsoft.com/en-us/p/microsoft-edge-devtools-preview/9mzbfrmz0mnj?activetab=pivot:overviewtab, Accessed: 2020-05-26.

[10] "webpack," https://webpack.js.org/, Accessed: 2020-05-27.

[11] "Cloud Speech-to-Text - Speech Recognition | Cloud Speech-to-Text | Google Cloud," https://cloud.google.com/speech-to-text/, Accessed: 2019-11-08.