Criterion C: Development

__main__.py

Modules included:

To make my interpreter application, I'm going to use external python modules to manage some of the underlying complexity that my app requires. Those are:

- SpeechRecognition: speech-to-text (STT)
- Googletrans: translation
- gTTS: text-to-speech (TTS)
- GTK 3: for the graphical user interface (GUI)
    - To make the GUI, I'm going to use Glade, a graphical GUI designer application.

```
# General
import os
import time
from threading import Thread

# External modules
from googletrans import Translator
import speech_recognition as sr
from gtts import gTTS
from playsound import playsound

# Interface
import gi
gi.require_version("Gtk", "3.0")
gi.require_version("Gdk", "3.0")
from gi.repository import Gtk, Gdk, Gio

# Application
from interpreter import Interpreter
from handler import Handler
import dictionaries as dic
```

Body:

The main program file does the following: creates a builder instance that reads the widgets (objects) of the interface described in an XML file, then joins signal-response pairs managed by the handler, after this it loads the CSS theme, joins the event of closing the tab to exiting the program and finally starts the main loop.

```
# The Builder class creates the interface objects (widgets) from file
builder = Gtk.Builder()
builder.add_from_file("interface/gui.glade")

# Join signal-response pairs managed by the Handler class
```

```
handler = Handler(builder)
builder.connect_signals(handler)

# Load CSS theme
screen = Gdk.Screen.get_default()
provider = Gtk.CssProvider()
provider.load_from_path("/home/amaldok/Prog/CS-IA/interface/theme.css") # full path needed
Gtk.StyleContext.add_provider_for_screen(screen, provider,
Gtk.STYLE_PROVIDER_PRIORITY_APPLICATION)

# Load main window
window = builder.get_object("MainWindow")
window.connect("destroy", Gtk.main_quit) # Join quit signal to destroy main window
window.show_all()                        # Show all widgets

# Start the main loop
Gtk.main()
```

interpreter.py

Constructor:

The constructor takes two parameters that are handler methods. For the attributes, the default values are given as well as initializing the lists in which the translation history will be stored. Also, it creates an instance of the recognizer and the translator of the speechrecognition and googletrans modules respectively. The translator doesn't need further settings, but for the recognizer we need to indicate the pause_threshold, the time in seconds that we consider to be a pause in speech (to cut speech into pieces), and the microphone that we want to use as input device, that is set to the default.

```
class Interpreter():
    "Translates, reproduces sound, stores translation history"

    def __init__(self, display_spoken_text, display_translated_text):
        self.status = "paused"
        self.path = os.getcwd()             # Full path to temporarily store audio files

        self.src_lang = "es"                # Set defaults
        self.dest_lang = "en"
        self.transcribed_texts = []         # Lists for translations history
        self.translated_texts = []
        self.trans_texts_lang = []

        self.recognizer = sr.Recognizer()   # Recognizer instance
        self.recognizer.pause_threshold = 1 # Set time (in seconds) of a pause in speech

        # From microphone input devices store index of default
        device_list = sr.Microphone.list_microphone_names()
```

```
        self.input_idx = device_list.index("default")

        self.translator = Translator()      # Translator instance

        # Callbacks to Handler to display text once it is available
        self.spoken_text_callback = display_spoken_text
        self.translated_text_callback = display_translated_text
```

Play/pause:

The start method initiates the interpreting process by changing the status and starting a thread of the recognize method. (This will be further explained later on.) For the stop method, it simply changes the status to notify running processes to stop.

```
def start(self):
    self.status = "interpreting"
    t = Thread(target=self.interpreter_recognize)
    t.start()

def stop(self):
    self.status = "paused"
```

Language setters:

```
def set_src_lang(self, lang="es"):
    self.src_lang = lang

def set_dest_lang(self, lang="en"):
    self.dest_lang = lang
```

Translation history getters:

```
def get_size(self):
    return len(self.transcribed_texts)

def get_transcribed_text(self, idx):
    return self.transcribed_texts[idx]

def get_translated_text(self, idx):
    return self.translated_texts[idx]

def get_trans_text_lang(self, idx):
    return self.trans_texts_lang[idx]
```

Recognize:

The recognize method needs to run always as long as the interpreter isn't paused. If we recognize a sentence, translate it and wait for it to be reproduced audibly, by that time the speaker could have spoken more and the program wouldn't have been listening. That is why the

interpreting process cannot be conceived as a linear one, we need to do multiple things at the same time.
That's why I am using multithreading in this part of the program. It is a technique that allows multiple processes to run simultaneously, that is exactly what the interpreting process needs. Each fragment of speech that the recognizer records is executed in a separate thread that also translates and reproduces the translated speech by calling two methods that are found below. But before it calls those two processes, it spawns a new thread to keep listening, that is, the recognize method is called in this new thread. That way, I can ensure that the program is listening all the time. Because as soon as a sentence has been recorded, before any further processes, a new thread is spawned to continue listening.

```
def interpreter_recognize(self):
    if self.status == "paused":
        return

    with sr.Microphone(sample_rate=44100, device_index=self.input_idx) as source:
        try:
            # Record a phrase
            audio = self.recognizer.listen(source, timeout=10, phrase_time_limit=10)
            # Recognize speech
            text = self.recognizer.recognize_google(audio, language=self.src_lang)

        except sr.WaitTimeoutError: # User has not spoken within limit time
            print("WaitTimeoutError")
            t = Thread(target=self.interpreter_recognize)
            t.start() # Start new thread

        except sr.UnknownValueError: # Speech is unrecognizable
            print("UnknownValueError")
            t = Thread(target=self.interpreter_recognize)
            t.start() # Start new thread

        else:
            self.transcribed_texts.append(text)     # Update translation history
            self.spoken_text_callback(text, True)   # Display transcribed text
            t = Thread(target=self.interpreter_recognize)
            t.start()                               # Start new thread
            self.interpreter_translate(text)        # Continue interpretation process ->
translate
```

Translate:

Apart from translating using the translator instance, the translation and its language is stored in the lists of the translation history in order for them to be retrieved later if needed. Then the final stage of the interpreting process, to reproduce the speech, is called.

```
def interpreter_translate(self, text):
    # Translate
```

```
    translation = self.translator.translate(text, dest=self.dest_lang, src=self.src_lang)
    text = translation.text

    # Update translation history
    self.translated_texts.append(text)
    self.trans_texts_lang.append(self.dest_lang)
    self.translated_text_callback(text) # Display translated text

    self.interpreter_reproduce(text, self.dest_lang) # Reproduce translated text as audio
```

Reproduce:

```
def interpreter_reproduce(self, text, lang):
    tts = gTTS(text, lang=lang)          # Get audio
    tts.save(self.path + "/temp.mp3")    # Store temporarily
    playsound(self.path + "/temp.mp3")   # Play
    os.remove(self.path + "/temp.mp3")   # Erase file
```

handler.py

Constructor:

The handler constructor takes as parameter the builder that has already read the XML file, in order to access the widgets through it. It stores the index of the current translation displayed (-1 if none) that is also used to navigate through the translation history. Then it fills the language chooser menus and sets active the default languages (same as interpreter defaults). Then, for some type of incompatibility, the widgets that are overridden with a CSS theme needed to be renamed in order for CSS to recognize them (I have renamed them with their exact previous name). Finally, the arrows are set insensitive since the translation history is still empty and navigation is pointless.

```
class Handler():
    "Handles events (signals) triggered in the GUI by the user"

    def __init__(self, builder):
        # Create instance of Interpreter passing two Handler methods as callbacks
        self.interpreter = Interpreter(self.display_spoken_text,
self.display_translated_text)
        self.builder = builder
        self.current_idx = -1 # Index of the translations array (stored in interpreter)

        # Fill comboboxes with language names
        src = self.builder.get_object("SrcLangComboBox")
        dest = self.builder.get_object("DestLangComboBox")
        for i in range(4):
            src.append_text(dic.languages[i])
            dest.append_text(dic.languages[i])
        # Set defaults
```

```
        src.set_active(1)
        dest.set_active(0)

        # Rename widgets that have CSS overriden theme (otherwise CSS does not recognize
them)
        for name in cssList:
            widget = self.builder.get_object(name)
            widget.set_name(name) # Assign same name

        # Set arrows insensitive
        rArrow = self.builder.get_object("RightArrowButton")
        rArrow.set_sensitive(False)
        lArrow = self.builder.get_object("LeftArrowButton")
        lArrow.set_sensitive(False)
```

Play/pause button:

When the interpreter is running, it is useful to set some of the elements of the interface insensitive, that is to make them not responsible, in order to avoid scenarios where the user changes, for example, the source language while translating.

```
def on_toggled_button(self, button):
    active = button.get_active() # Get new status

    # Retrieve widgets to modify
    label = self.builder.get_object("PlayPauseLabel")
    image = self.builder.get_object("PlayPauseImage")
    src = self.builder.get_object("SrcLangComboBox")
    dest = self.builder.get_object("DestLangComboBox")
    switch = self.builder.get_object("SwitchButton")
    sound = self.builder.get_object("SoundButton")
    copy = self.builder.get_object("CopyClipboard")

    if active:
        label.set_text("Pause")
        image.set_from_icon_name("media-playback-pause", 2)
        self.interpreter.start() # Start interpreting

        # Set insensitive
        src.set_sensitive(False)
        dest.set_sensitive(False)
        switch.set_sensitive(False)
        sound.set_sensitive(False)
        copy.set_sensitive(False)

    else:
        label.set_text("Play")
        image.set_from_icon_name("media-playback-start", 2)
        self.interpreter.stop() # Stop interpreting
```

```
        # Set sensitive
        src.set_sensitive(True)
        dest.set_sensitive(True)
        switch.set_sensitive(True)
        sound.set_sensitive(True)
        copy.set_sensitive(True)
```

Change languages:

```
def on_changed_src_lang(self, comboBox):
    lang = comboBox.get_active_text()
    code = dic.lang_to_code[lang]
    self.interpreter.set_src_lang(code) # Change src on Interpreter

def on_changed_dest_lang(self, comboBox):
    lang = comboBox.get_active_text()
    code = dic.lang_to_code[lang]
    self.interpreter.set_dest_lang(code) # Change dest on Interpreter

def on_clicked_switch_button(self, button):
    src = self.builder.get_object("SrcLangComboBox")
    dest = self.builder.get_object("DestLangComboBox")

    # Retrieve language and index
    srcIdx = src.get_active()
    srcText = src.get_active_text()
    destIdx = dest.get_active()
    destText = dest.get_active_text()

    # Perform swap
    src.set_active(destIdx)
    dest.set_active(srcIdx)
    self.interpreter.set_src_lang(dic.lang_to_code[destText])
    self.interpreter.set_dest_lang(dic.lang_to_code[srcText])
```

Copy to clipboard:

```
def on_clicked_copyclipboard(self, button):
    if self.current_idx != -1:
        text = self.interpreter.get_translated_text(self.current_idx)
        clipboard = Gtk.Clipboard.get(Gdk.SELECTION_CLIPBOARD)
        clipboard.set_text(text, len(text)) # Set text on clipboard
```

Sound button:

The workings of this button is very simple: it retrieves the text by the index and then reproduces it by calling the reproduce method of the translator.

```
def on_soundbutton_clicked(self, button):
    if self.current_idx != -1:
        # Get text and language at idx (the one at display)
        text = self.interpreter.get_translated_text(self.current_idx)
        lang = self.interpreter.get_trans_text_lang(self.current_idx)
        self.interpreter.interpreter_reproduce(text, lang) # Call Interpreter reproduce
method
```

History navigation:

Two methods to navigate through the history of translations: when the right or left arrows are pressed (move back and forth). The two conditionals in each check that we are on the first/last elements of the history and sets the arrows insensitive to avoid moving beyond bounds. The translation that needs to be displayed is fetched using the proper index and the interpreter getter. Then, the proper handler methods to display text on screen are called.

```
def on_rightarrow_clicked(self, button):
    sz = self.interpreter.get_size()
    if (self.current_idx + 1 == sz): # Out of bounds
        return

    self.current_idx += 1
    lArrow = self.builder.get_object("LeftArrowButton")
    lArrow.set_sensitive(True)
    if (self.current_idx + 1 == sz): # If last, set right unsensitive
        rArrow = self.builder.get_object("RightArrowButton")
        rArrow.set_sensitive(False)

    # Get text by index in the lists stored in Interpreter and display
    srcText = self.interpreter.get_transcribed_text(self.current_idx)
    self.display_spoken_text(srcText, False)
    destText = self.interpreter.get_translated_text(self.current_idx)
    self.display_translated_text(destText)

def on_leftarrow_clicked(self, button):
    sz = self.interpreter.get_size()
    if (self.current_idx - 1 == -1): # Out of bounds
        return

    self.current_idx -= 1
    rArrow = self.builder.get_object("RightArrowButton")
    rArrow.set_sensitive(True)
    if (self.current_idx == 0): # If first, set left unsentive
        lArrow = self.builder.get_object("LeftArrowButton")
        lArrow.set_sensitive(False)

    # Get text by index in the lists stored in Interpreter and display
    srcText = self.interpreter.get_transcribed_text(self.current_idx)
    self.display_spoken_text(srcText, False)
```

```
    destText = self.interpreter.get_translated_text(self.current_idx)
    self.display_translated_text(destText)
```

Display text:

These two methods are different from the previous ones, since they are not triggered by a signal the user produces on the interface. When the displayed text must be changed, they are called from inside the class by the history navigation methods or by the interpreter instance as callbacks.
A callback is a function passed as an argument of another function. In this case, these two functions are passed to the interpreter instance when it is constructed, and the interpreter stores these two functions as attributes. Once the interpreter has reached a point in its translation process that needs to display some text on the interface, it calls either of these functions depending on the case and leaves the interface management to the handler.
I have decided to use callbacks in my implementation because they helped me to maintain the interpreter exclusively in the backend (translation), while things such as displaying text are left to the GUI manager that is the handler.

```
def display_spoken_text(self, text, newSentence):
    srcBuffer = Gtk.TextBuffer()                            # Create new buffer
    i = srcBuffer.get_start_iter()
    srcBuffer.do_insert_text(srcBuffer, i, text, len(text)) # Put text on buffer
    srcTextView = self.builder.get_object("SpokenTextView")
    srcTextView.set_buffer(srcBuffer)                       # Put buffer on text view

    if newSentence: # Whether it's the last sentence translated by Interpreter (when used
as callback)
        self.current_idx = self.interpreter.get_size()-1    # Set idx to last
        rArrow = self.builder.get_object("RightArrowButton")
        rArrow.set_sensitive(False)
        if (self.current_idx > 0): # If there are previous sentences, allow to navigate
back
            lArrow = self.builder.get_object("LeftArrowButton")
            lArrow.set_sensitive(True)

def display_translated_text(self, text):
    destBuffer = Gtk.TextBuffer()                               # Create new buffer
    i = destBuffer.get_start_iter()
    destBuffer.do_insert_text(destBuffer, i, text, len(text))   # Put text on buffer
    destTextView = self.builder.get_object("TranslatedTextView")
    destTextView.set_buffer(destBuffer)                         # Put buffer on text view
```

Additional files

theme.css

This small chunk of CSS simply overrides the buttons default appearance to make the borders of two adjacent buttons (the arrow buttons) sharp in order for them to look more visually appealing.

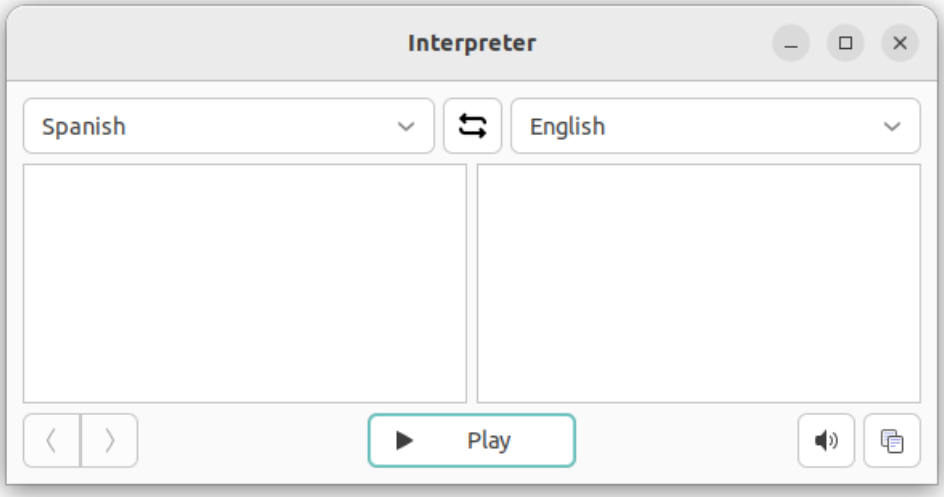
```
button#RightArrowButton {
    border-top-left-radius: 0;
    border-bottom-left-radius: 0;
}

button#LeftArrowButton {
    border-top-right-radius: 0;
    border-bottom-right-radius: 0;
}
```

dictionaries.py

```
# For portraying purposes. More languages can be added

code_to_lang = {"en": "English",
                "es": "Spanish",
                "fr": "French",
                "de": "German"}

lang_to_code = {"English": "en",
                "Spanish": "es",
                "French": "fr",
                "German": "de"}

languages = ["English", "Spanish", "French", "German"]
```

Interface

This is the final appearance of the interface:



Wordcount: 947