

AccFileOpenner.py

```
import os
import re
import tkinter as tk
from tkinter import messagebox, filedialog
import shutil
import sys
import time
from gui import BaseApp
from tkinterdnd2 import TkinterDnD
import subprocess

sys.path.append(os.path.dirname(os.path.dirname(__file__)))
import _Exe_Util

FILE_CATALOG = {
    "indesign": {
        "lock_file_extension": ".idlk",
        "lock_file_begin_template": "~{}",
        "file_extension": ".indd",
    },
    "rhino": {
        "lock_file_extension": ".rhl",
        "lock_file_begin_template": "{}",
        "file_extension": ".3dm",
    },
    "cad": {
        "lock_file_extension": ".dwl",
        "lock_file_begin_template": "{}",
        "file_extension": ".dwg",
    },
    "pdf": {
        "lock_file_extension": None,
        "lock_file_begin_template": "",
        "file_extension": ".pdf",
    },
    "excel": {
        "lock_file_extension": ".xlsx",
        "lock_file_begin_template": "~${}",
        "file_extension": ".xlsx",
        "prefer_cloud": True
    },
    "word": {
        "lock_file_extension": ".docx",
        "lock_file_begin_template": "~${}",
        "file_extension": ".docx",
        "prefer_cloud": True
    },
    "photoshop": {
        "lock_file_extension": None,
        "lock_file_begin_template": "",
        "file_extension": ".psd",
    },
    "illustrator": {
        "lock_file_extension": None,
        "lock_file_begin_template": "",
        "file_extension": ".ai",
    },
    "grasshopper": {
        "lock_file_extension": None,
        "lock_file_begin_template": "",
        "file_extension": ".gh",
    },
}

class FileProcessorApp(BaseApp):
    def __init__(self, root):
```

```

self.username = _Exe_Util.get_username()
super().__init__(root)
self.selected_file = ""
self.original_file = None

possible_acc_folders = [
f"{os.getenv('USERPROFILE')}\\DC\\ACCDocs",
f"{os.getenv('USERPROFILE')}\\ACCDocs"
]
for acc_folder in possible_acc_folders:
    if os.path.exists(acc_folder):
        self.acc_folder = acc_folder
        break

self.lock_file = None
self.finished_button = None
self.monitor_acc_folder()

def monitor_acc_folder(self):
    self.update_editing_and_requesting_files()
    self.root.after(2000, self.monitor_acc_folder)

def handle_file_selection(self, file_path):
    if self.original_file:
        response = messagebox.showinfo(
            "Job in Progress",
            "You already have a file being processed. To monitor another
file, start a new AccFileOpener."
        )
        return

    if file_path:
        self.process_file(file_path)

def open_file_dialog(self, event=None):
    file_path = filedialog.askopenfilename()
    self.handle_file_selection(file_path)

def handle_file_drop(self, event):
    file_path = event.data.strip("{}")
    print(f"File dropped: {file_path}")
    self.handle_file_selection(file_path)

@_Exe_Util.try_catch_error
def process_file(self, file_path):
    self.original_file = file_path
    print(f"Processing file: {self.original_file}")

    file_category_data = self.get_file_category_data()
    if file_category_data.get("prefer_cloud", False):
        response = messagebox.askyesno(
            "Cloud Editing Preferred",
            "This file type is preferred to be edited on the ACC cloud for
realtime collaboration. Do you still want to open it locally?",
            icon=messagebox.QUESTION,
            default=messagebox.NO
        )
        if not response:
            messagebox.showinfo("Cloud Editing", "Right click on the file in
Windows Explorer and click 'View Online'.")
            self.reset_all()
            return

    if self.check_self_editing():
        messagebox.showwarning("Warning", "You are already editing this
file. Will not attempt to open twice.\nIf you have recently crashed the
AccFileOpener, remove that editing marker file and retry.")
        self.reset_all()
        return

```

```

self.update_file_path_label()
self.copy_file_to_desktop()
os.startfile(self.desktop_file)

self.cleanup_old_request_files()
current_editor = self.check_existing_editors()
if current_editor:
    self.create_request_file()
    messagebox.showwarning("Warning", f"This file is currently edited by
{current_editor}.\n\nA request file has been placed.")
    return

self.create_editing_marker()
self.monitor_file_lock()

def copy_file_to_desktop(self):
    file_name = os.path.basename(self.original_file)
    desktop_path = os.path.join(os.path.join(os.environ['USERPROFILE']),
'Desktop')
    self.desktop_file = os.path.join(desktop_path, file_name)
    shutil.copyfile(self.original_file, self.desktop_file)

def get_lock_file(self):
    file_category_data = self.get_file_category_data()
    lock_file_extension = file_category_data["lock_file_extension"]
    lock_file_begin_template =
file_category_data["lock_file_begin_template"]
    base_name = os.path.basename(self.desktop_file)
    lock_file_begin = lock_file_begin_template.format(base_name.replace(file
_category_data["file_extension"], ""))

    # word lock file sometimes remove first two char to create lockfile, so
need to search as well
    lock_file_begin_alt = lock_file_begin_template.format(base_name[2:].repl
ace(file_category_data["file_extension"], ""))

    if lock_file_extension is None:
self.create_finished_button(file_category_data["file_extension"].strip('.'))
    return

    for f in os.listdir(os.path.dirname(self.desktop_file)):
        if f.endswith(lock_file_extension) and
(f.lower().startswith(lock_file_begin.lower()) or
f.lower().startswith(lock_file_begin_alt.lower())):
            print(f"File lock found: {f}")
            self.lock_file =
os.path.join(os.path.dirname(self.desktop_file), f)
            return

    print("File lock not found")

def monitor_file_lock(self):
    if not self.lock_file and not self.finished_button:
        self.get_lock_file()
        self.root.after(1000, self.monitor_file_lock)
        return

    if self.finished_button:
        print("Finished button displayed, waiting for user action.")
        return

    if not os.path.exists(self.lock_file):
        print("Lock file gone, ready to sync back to ACC")
        self.copy_back_to_original()
        print("Job done!")
    else:
        print("Lock file still exists, user is still editing")
        self.root.after(1000, self.monitor_file_lock)

def copy_back_to_original(self):
    if self.original_file and self.desktop_file:

```

```

        print(f"Copying back to original: {self.desktop_file} to
{self.original_file}")
        shutil.copy2(self.desktop_file, self.original_file)
        os.remove(self.desktop_file)
        self.remove_editing_marker()
        self.reset_all()

    def clear_file_path_label(self):
        self.file_path_label.config(text="")

    def update_file_path_label(self):
        wrap_length = int(self.windowX * 0.8)
        self.file_path_label.config(text=self.original_file,
wraplength=wrap_length)

    def cleanup_old_request_files(self):
        file_name = os.path.basename(self.original_file)
        dir_name = os.path.dirname(self.original_file)
        print(f"Cleaning up old request files in {dir_name}")
        for file in os.listdir(dir_name):
            if re.match(rf'\[{self.username}_requesting\]_{file_name}', file):
                print(f"Removing request file: {file}")
                os.remove(os.path.join(dir_name, file))

    def check_existing_editors(self):
        file_name = os.path.basename(self.original_file)
        dir_name = os.path.dirname(self.original_file)
        print(f"Checking for existing editors in {dir_name}")
        for file in os.listdir(dir_name):
            search = re.match(rf'\[(\w+)_editing\]_{file_name}', file)
            if search:
                username = search.group(1)
                print(f"File is currently being edited by: {username}")
                return username
        return None

    def check_self_editing(self):
        file_name = os.path.basename(self.original_file)
        dir_name = os.path.dirname(self.original_file)
        for file in os.listdir(dir_name):
            if re.match(rf'\[{self.username}_editing\]_{file_name}', file):
                return True
        return False

    def create_editing_marker(self):
        file_name = os.path.basename(self.original_file)
        marker_file = os.path.join(os.path.dirname(self.original_file),
f"[{self.username}_editing]_{file_name}")
        with open(marker_file, "w") as f:
            f.write(f"This file is currently being edited by {self.username}.")
        print(f"Created editing marker: {marker_file}")

    def get_file_category_data(self):
        file_extension = os.path.splitext(self.original_file)[1].lower()
        for category, info in FILE_CATALOG.items():
            if info["file_extension"] == file_extension:
                return FILE_CATALOG[category]
        return None

    def remove_editing_marker(self):
        if not self.original_file:
            return
        marker_file = os.path.join(os.path.dirname(self.original_file),
f"[{self.username}_editing]_{os.path.basename(self.original_file)}")

    def try_remove_marker():
        if os.path.exists(marker_file):
            try:
                os.remove(marker_file)
                print("Editing marker removed")
            except Exception as e:
                print(f"Error removing editing marker: {e}")

```

```

        self.root.after(1000, try_remove_marker)

    try_remove_marker()

    def create_request_file(self):
        request_file_name =
f"[{self.username}_requesting]_{os.path.basename(self.original_file)}"
        request_file_path = os.path.join(os.path.dirname(self.original_file),
request_file_name)
        with open(request_file_path, "w") as f:
            f.write(f"Request to edit the file by {self.username}")
            print(f"Created request file: {request_file_path}")

    def check_request_files(self):
        request_users = []
        dir_name = os.path.dirname(self.original_file)
        for file in os.listdir(dir_name):
            if
re.match(rf'[(\w+)_requesting\_]{os.path.basename(self.original_file)}', file):
                request_users.append(file.split('_')[0][1:])
        return request_users

    def update_editing_and_requesting_files(self):

        editing_files = []
        requesting_files = []
        for root, dirs, files in os.walk(self.acc_folder):
            if "_D" in root or "_C" in root:
                continue
            for file in files:
                if re.match(r'[(\w+_editing\)]', file):
                    editing_files.append(os.path.join(root, file))
                elif re.match(r'[(\w+_requesting\)]', file):
                    requesting_files.append(os.path.join(root, file))
            self.update_editing_files_panel(editing_files, requesting_files)

    def update_editing_files_panel(self, editing_files, requesting_files):
        self.editing_files_text.configure(state='normal')
        self.editing_files_text.delete('1.0', tk.END)
        self.editing_files_text.tag_configure('header', font=('Helvetica', 12,
'bold'))
        self.editing_files_text.tag_configure('body', font=('Helvetica', 8,
'normal'))

        self.editing_files_text.insert(tk.END, "Editing Files:\n", 'header')
        for file in editing_files:
            self.insert_clickable_file(file)

        self.editing_files_text.insert(tk.END, "\nRequesting Files:\n",
'header')
        for file in requesting_files:
            self.insert_clickable_file(file)

        self.editing_files_text.configure(state='disabled')

    def insert_clickable_file(self, file_path):
        start_index = self.editing_files_text.index(tk.END)
        self.editing_files_text.insert(tk.END, f"{file_path}\n", 'body')
        end_index = self.editing_files_text.index(tk.END)
        tag_name = f"tag_{file_path}" # Use a unique tag name for each file
path
        self.editing_files_text.tag_add(tag_name, start_index, end_index)
        self.editing_files_text.tag_bind(tag_name, "<Button-1>", lambda e,
path=file_path: self.open_file_folder(path))
        self.editing_files_text.tag_bind(tag_name, "<Enter>", lambda e,
tag=tag_name: self.on_enter(e, tag))
        self.editing_files_text.tag_bind(tag_name, "<Leave>", lambda e,
tag=tag_name: self.on_leave(e, tag))

    def on_enter(self, event, tag):
        event.widget.config(cursor="cross")
        event.widget.tag_configure(tag, background="yellow", foreground="black")

```

```

def on_leave(self, event, tag):
    event.widget.config(cursor="")
    event.widget.tag_configure(tag, background="", foreground="")

def open_file_folder(self, file_path):
    folder_path = os.path.dirname(file_path)
    if os.path.exists(folder_path):
        os.startfile(folder_path)
    else:
        messagebox.showerror("Error", "Folder not found.")

def create_finished_button(self, file_type):
    if self.finished_button:
        self.finished_button.destroy()
    self.finished_button = tk.Button(self.root,
                                     text=f"I am finished with this
[file_type] file, click to remove the [editing] marker file.",
                                     command=self.copy_back_to_original)
    self.finished_button.grid(row=4, column=0, columnspan=3, pady=10)

def remove_finished_button(self):
    if self.finished_button:
        self.finished_button.destroy()
        self.finished_button = None

def reset_all(self):
    self.original_file = None
    self.selected_file = ""
    self.lock_file = None
    self.finished_button = None
    self.clear_file_path_label()

@_Exe_Util.try_catch_error
def main():
    root = TkinterDnD.Tk()
    app = FileProcessorApp(root)
    root.mainloop()

if __name__ == "__main__":
    main()

```

gui.py

```

import os
import tkinter as tk
from tkinter import scrolledtext
from PIL import Image, ImageTk
import datetime
from tkinterdnd2 import DND_FILES
import sys
sys.path.append(os.path.dirname(os.path.dirname(__file__)))
import _GUI_Base_Util
EXPIRATION_DATE = datetime.date(2025, 1, 1)

class BaseApp(_GUI_Base_Util.BaseGUI):
    def __init__(self, root):
        self.root = root
        self.root.configure(bg=self.BACKGROUND_COLOR_HEX)
        self.root.geometry("1000x800")
        self.setup_icon()
        self.create_widgets()
        self.setup_bindings()
        self.update_title_with_days_left()
        self.create_dashboard()
    def setup_icon(self):

```

```

        icon_path = os.path.join(os.path.dirname(__file__), "icon_ennead-e.ico")
        self.root.iconbitmap(icon_path)
        self.logo_path = os.path.join(os.path.dirname(__file__), "logo.png")
        self.logo_image = Image.open(self.logo_path)
        self.logo_image = self.logo_image.resize((self.logo_image.width // 2,
self.logo_image.height // 2), Image.LANCZOS)
        self.logo_photo = ImageTk.PhotoImage(self.logo_image)

    def create_widgets(self):
        self.logo_label = tk.Label(self.root, image=self.logo_photo,
bg=self.BACKGROUND_COLOR_HEX)
        self.logo_label.grid(row=0, column=0, columnspan=3, sticky="nsew")

        self.editing_files_frame = tk.Frame(self.root,
bg=self.BACKGROUND_COLOR_HEX)
        self.editing_files_frame.grid(row=1, column=0, columnspan=3,
sticky="nsew")

        self.editing_files_text =
scrolledtext.ScrolledText(self.editing_files_frame, width=60, height=15,
bg=self.BACKGROUND_COLOR_HEX, fg='white', font=('Helvetica', 12, 'bold'),
wrap=tk.WORD)
        self.editing_files_text.pack(fill=tk.BOTH, expand=True)
        self.editing_files_text.config(height=10)
        self.editing_files_text.config(height=10) # Setting fixed height

        note = "The file will open automatically after picked/dropped."
        note += "\nAccepting File types of Indesign, Rhino, Word, Excel, PDF,
Photoshop, Illustrator"
        self.instructions_label = tk.Label(self.root, text=note,
bg=self.BACKGROUND_COLOR_HEX, fg='white', font=('Helvetica', 12),
wraplength=800, justify=tk.LEFT)
        self.instructions_label.grid(row=2, column=0, columnspan=3, sticky="nw",
padx=20, pady=10)

        for i in range(3):
            self.root.grid_columnconfigure(i, weight=1)
        self.root.grid_rowconfigure(1, weight=1)

    def setup_bindings(self):
        self.root.bind("<Motion>", self.rotate_logo)

    def rotate_logo(self, event):
        max_rotation = 20
        width = self.root.winfo_width()
        relative_x = event.x / width
        angle = (relative_x * 2 - 1) * max_rotation
        angle = max(min(angle, max_rotation), -max_rotation)
        rotated_image = self.logo_image.rotate(angle)
        self.logo_photo = ImageTk.PhotoImage(rotated_image)
        self.logo_label.configure(image=self.logo_photo)

    def create_dashboard(self):
        self.dashboard_frame = tk.Frame(self.root, bg='#3e3e3e', height=100)
        self.dashboard_frame.grid(row=3, column=0, columnspan=3, sticky="nsew")
        self.root.grid_rowconfigure(3, weight=1)

        self.canvas = tk.Canvas(self.dashboard_frame, bg='#3e3e3e',
highlightthickness=0)
        self.canvas.pack(fill=tk.BOTH, expand=True, padx=20, pady=20)

        self.root.update_idletasks() # Ensure the window is fully rendered
        self.windowX = self.canvas.winfo_width()
        self.windowY = self.canvas.winfo_height()

        self.draw_rounded_rect(10, 10, self.windowX - 10, self.windowY - 10, 20,
width=4, dash=(5, 3)) # Adjusted the bottom padding

        self.dashboard_label = tk.Label(self.canvas, text="Drag and Drop a file
here or Click to Select a file", bg='#3e3e3e', fg='white', font=('Helvetica',
14, 'bold'))

```

```

        self.dashboard_label.place(relx=0.5, rely=0.1, anchor='n') # Positioned
the label at the top

        self.file_path_label = tk.Label(self.canvas, text="", bg='#3e3e3e',
fg='white', font=('Helvetica', 12), wraplength=int(self.windowX * 0.8))
        self.file_path_label.place(relx=0.5, rely=0.5, anchor='center')

        self.dashboard_label.bind("<Button-1>", self.open_file_dialog)
        self.dashboard_frame.bind("<Button-1>", self.open_file_dialog)
        self.root.drop_target_register(DND_FILES)
        self.root.dnd_bind('<<Drop>>', self.handle_file_drop)

        self.dashboard_label.bind("<Enter>", self.change_cursor_to_hand)
        self.dashboard_label.bind("<Leave>", self.change_cursor_to_arrow)
        self.dashboard_frame.bind("<Enter>", self.change_cursor_to_hand)
        self.dashboard_frame.bind("<Leave>", self.change_cursor_to_arrow)

def change_cursor_to_hand(self, event):
    return
    event.widget.config(cursor="hand2")

def change_cursor_to_arrow(self, event):
    return
    event.widget.config(cursor="")

def draw_rounded_rect(self, x1, y1, x2, y2, radius, **kwargs):
    # Draw the lines
    self.canvas.create_line(x1 + radius, y1, x2 - radius, y1, **kwargs)
    self.canvas.create_line(x2, y1 + radius, x2, y2 - radius, **kwargs)
    self.canvas.create_line(x2 - radius, y2, x1 + radius, y2, **kwargs)
    self.canvas.create_line(x1, y2 - radius, x1, y1 + radius, **kwargs)

    # Draw the arcs
    self.canvas.create_arc(x1, y1, x1 + 2 * radius, y1 + 2 * radius,
start=90, extent=90, style='arc', **kwargs)
    self.canvas.create_arc(x2 - 2 * radius, y1, x2, y1 + 2 * radius,
start=0, extent=90, style='arc', **kwargs)
    self.canvas.create_arc(x2 - 2 * radius, y2 - 2 * radius, x2, y2,
start=270, extent=90, style='arc', **kwargs)
    self.canvas.create_arc(x1, y2 - 2 * radius, x1 + 2 * radius, y2,
start=180, extent=90, style='arc', **kwargs)

def update_title_with_days_left(self):
    days_left = (EXPIRATION_DATE - datetime.date.today()).days
    if days_left <= 0:
        self.root.title("ACC File Opener - Tool Expired")
    elif days_left <= 30:
        self.root.title(f"ACC File Opener - {days_left} days left")
    else:
        self.root.title("ACC File Opener")

if __name__ == "__main__":
    root = tk.Tk()
    app = BaseApp(root)
    root.mainloop()

```