University School of Automation & Robotics
GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY
East Delhi Campus, Surajmal vihar
Delhi - 110092

Human Computer Interaction

ARD 320

Lab File

Artificial Intelligence & Machine Learning

(2022 - 2026)

Submitted by:
Priyankesh
11519011622
AIML B2

Submitted to:

Prof. Anupam Lakhanpal
USAR GGSIPU

S. No.	Topic	Page No.	Remarks	Signature
1.	To understand the trouble of interacting with machines - Redesign interfaces of home appliances like microwave oven, land-line phone, fully automatic washing machine.			
2.	To design and implement a calculator using both User-Centric and System-Centric approaches using the tkinter library in Python, and to observe the differences in usability, design principles, and user interaction.			
3.	Implementation of Menus for Graphical System which is to design and implement various types of menus using the Tkinter library to understand menu systems in user interfaces, including Menu Bar, Pull-Down, Cascading, and Pop-up menus.			
4.	Implementation of Different Kinds of Windows like Primary and Secondary Windows which is to design and implement a primary window and various types of secondary windows using Tkinter, demonstrating the flexibility and structure of graphical user interfaces.			
5.				
6.				
7.				
8.				

Experiment 1

Aim: To understand the trouble of interacting with machines - Redesign interfaces of home appliances like microwave oven, land-line phone, fully automatic washing machine.

- Objective 1: To understand the importance of human psychology in designing good interfaces.
- Objective 2: To encourage students to indulge into research in Machine Interface Design.
- Outcomes -
    - The end user will be able to apply HMI in their day – to – day activities.
    - The end user will be able to analyze the local and global impact of computing on individuals, organizations, and society.

Theory: Human-Computer Interaction (HCI) focuses on designing user-friendly systems by understanding how people interact with technology. A poor interface leads to user confusion, frustration, and errors. Good design considers:

- Human psychology
- User needs
- Ergonomics
- Usability principles

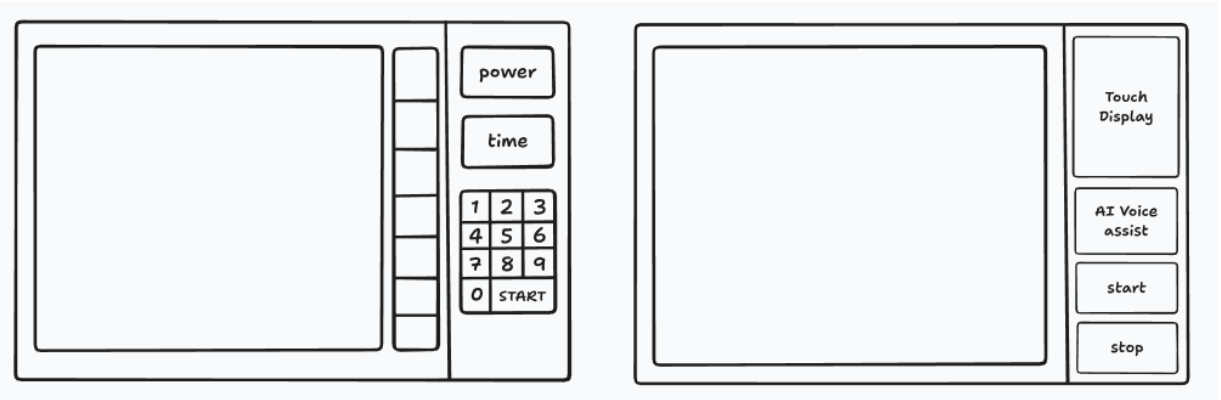
PACT-P Framework is used to evaluate and redesign interfaces:

- People – Who uses the system? Age, ability, background.
- Activities – What tasks are performed?
- Context – Where and under what conditions is it used?
- Technology – What interface mechanisms are involved?
- Process – How to improve or test the design.

Demo screens:

- Appliance 1: Microwave Oven

Existing Interface (Before) Redesigned Interface (After)

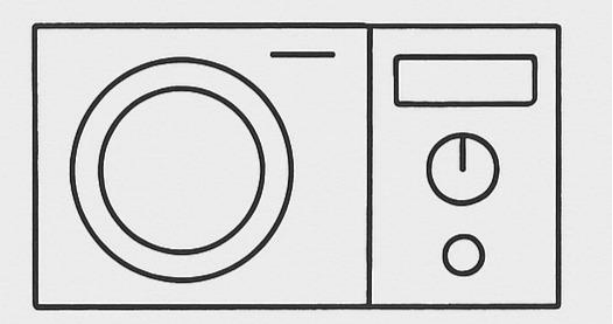


- Appliance 2: Landline Phone

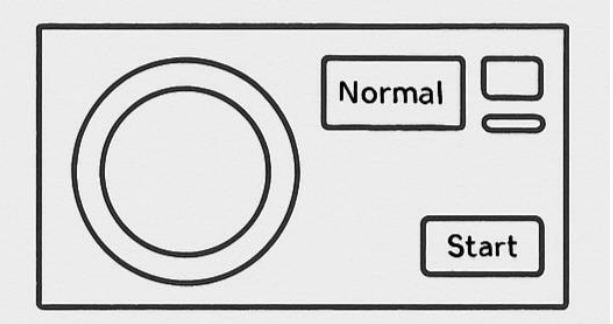
Existing Interface (Before) Redesigned Interface (After)

- Appliance 3: Washing Machine

Existing Interface (Before)

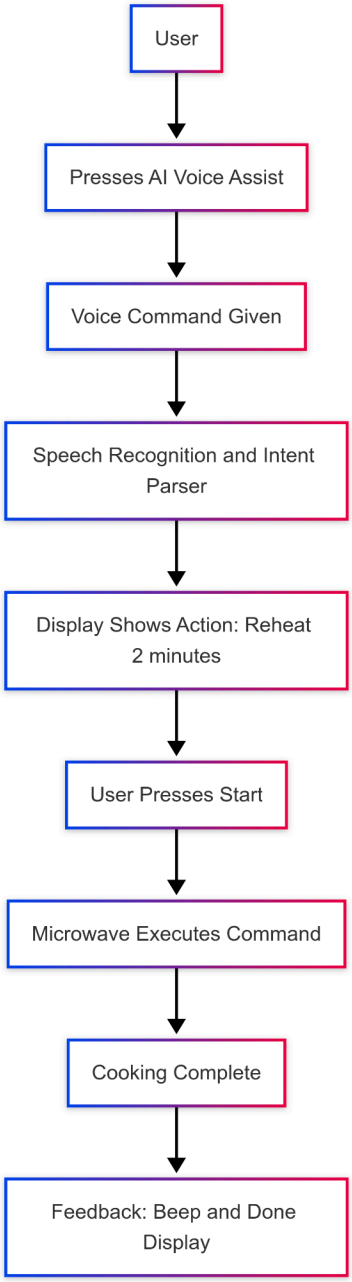


Redesigned Interface (After)

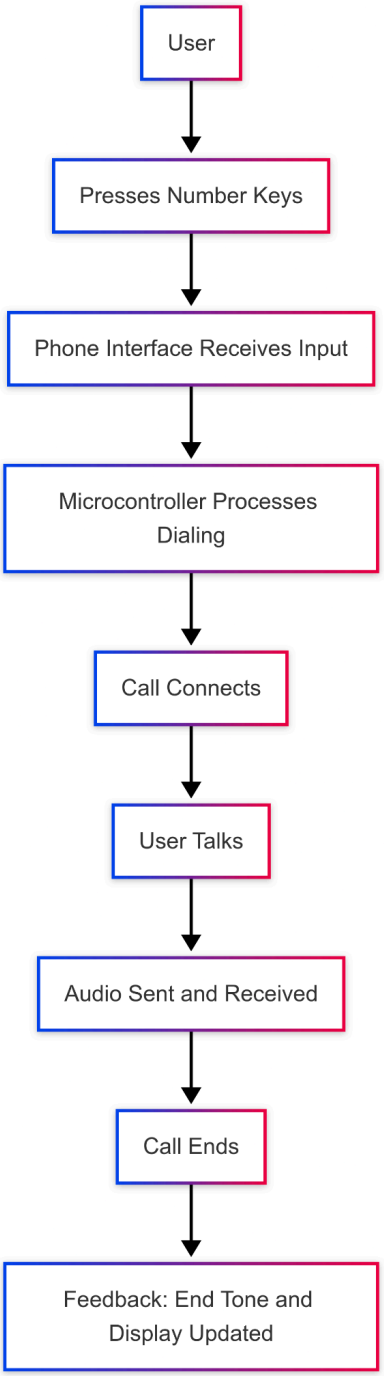


Working Flowcharts:

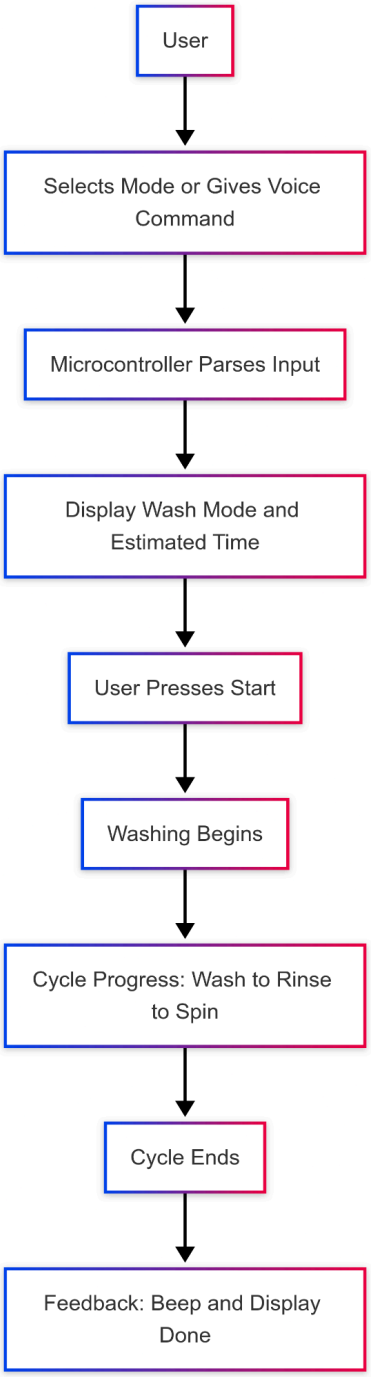
New Microwave



New Landline Phone



New Washing Machine



Observation:

1. Microwave Oven
    - Before Redesign:
        - Interface cluttered with too many unlabeled or confusing buttons.
        - No clear guidance for basic tasks like reheating or defrosting.
        - Small screen and poor feedback system.
        - Accessibility was low for elderly or tech-challenged users.
    - After Redesign:
        - Simplified with only three buttons: AI Voice Assist, Start, and Stop.
        - Users can give natural language commands (e.g., "Reheat for 2 minutes").
        - Clear display for command confirmation and cooking progress.
        - High accessibility and ease of use for all age groups.

2. Landline Phone
    - Before Redesign**:**
        - Small physical keypad with no display made dialing and accessing features difficult.
        - No caller ID or visual feedback.
        - Complex steps for common actions like redialing or accessing voicemail.
    - After Redesign:
        - Larger buttons and integrated screen improve usability and feedback.
        - Features like caller ID, call history, and speed dial simplify usage.
        - Interface supports quick, error-free calling, especially for elderly users.

3. Washing Machine
    - Before Redesign:
        - Overwhelming number of options with unlabeled icons.
        - No real-time feedback or cycle progress shown.
        - Users are often confused between similar-sounding wash modes.
    - After Redesign:
        - Simplified mode selection (e.g., Quick Wash, Regular, Delicates).
        - Voice command support optional for tech-savvy users.
        - Real-time display of remaining time and current stage (e.g., "Rinsing...").
        - Improved clarity and task efficiency with better visual and audible feedback.

Overall Observation:

Before redesign:

- Users were confused by too many buttons and unclear icons.
- Time settings required too many steps.
- Elderly users found the interface difficult to use.

After redesign:

- Users were able to complete tasks faster.
- Interface was easier to understand for all age groups.
- Reduced chances of errors.

Conclusion: This experiment helped us understand the importance of user-centered design. By analyzing interfaces using the PACT-P framework, we were able to redesign a more intuitive and efficient interface for a home appliance. This shows how HCI principles improve real-world usability and enhance user satisfaction.

Experiment 2

Aim: To design and implement a calculator using both User-Centric and System-Centric approaches using the `tkinter` library in Python, and to observe the differences in usability, design principles, and user interaction.

Theory:

- **Human-Computer Interaction (HCI):** HCI is the study and design of how users interact with computers and how to make this interaction effective, efficient, and satisfying.
- **User-Centric Design:** This approach focuses on the needs, preferences, and limitations of the end-user at every stage of the design process.
    - In this experiment:
        - For **ages 10-20**: A **fun, colorful, basic calculator** is shown to engage young users with simple functionality.
        - For **age >20**: A **professional, clean calculator** interface is shown for adults who prefer usability over visuals.
- **System-Centric Design:** This approach focuses on the system's functionality, optimization, and performance rather than the user's specific preferences.
    - In this experiment:
        - A **scientific calculator** is presented with optimized and advanced features like:
            - Logarithmic functions
            - Trigonometric operations
            - Exponentials, square roots, etc.
- **Differences between User-Centric and System-Centric Design:**

Feature	User-Centric	System-Centric
Focus	End-user needs and experience	System capabilities and efficiency
Flexibility	Adapts based on user characteristics	Same for all users
Example in this lab	Age-based interface	Advanced calculator with fixed UI
Complexity	Simplified and customized	Comprehensive and feature-rich

Code:

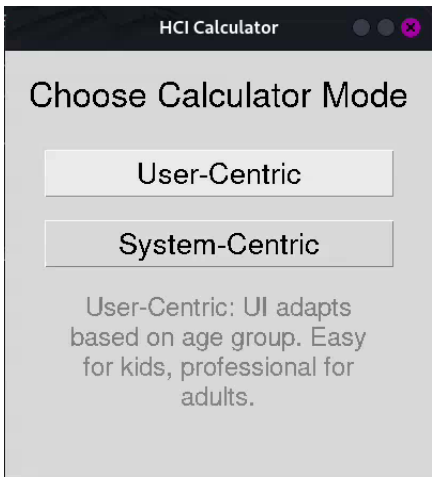
```
import tkinter as tk
from tkinter import messagebox
import math

class HCI_Calculator:
    def __init__(self, root):
        self.root = root
        self.root.title("HCI Calculator")
        self.root.geometry("400x400")
        self.main_menu()

    def main_menu(self):
        for widget in self.root.winfo_children():
            widget.destroy()
```

```
            self.info_label = tk.Label(self.root, text="Choose Calculator Mode",
font=("Arial", 24))
            self.info_label.pack(pady=20)

            self.user_btn = tk.Button(self.root, text="User-Centric", width=20,
command=self.user_mode, font=("Arial", 20))
            self.user_btn.pack(pady=10)

            self.system_btn = tk.Button(self.root, text="System-Centric", width=20,
command=self.system_mode, font=("Arial", 20))
            self.system_btn.pack(pady=10)

            self.hover_label = tk.Label(self.root, text="", wraplength=300,
fg="gray", font=("Arial", 18))
            self.hover_label.pack(pady=10)

            self.user_btn.bind("<Enter>", lambda e: self.hover_label.config(
                text="User-Centric: UI adapts based on age group. Easy for kids,
professional for adults."))
            self.user_btn.bind("<Leave>", lambda e:
self.hover_label.config(text=""))

            self.system_btn.bind("<Enter>", lambda e: self.hover_label.config(
                text="System-Centric: Full scientific calculator with optimized
functionality."))
            self.system_btn.bind("<Leave>", lambda e:
self.hover_label.config(text=""))

        def user_mode(self):
            self.clear_widgets()
            self.age_label = tk.Label(self.root, text="Enter your age:",
font=("Arial", 20))
            self.age_label.pack(pady=10)

            self.age_entry = tk.Entry(self.root, font=("Arial", 18))
            self.age_entry.pack(pady=5)

            self.submit_btn = tk.Button(self.root, text="Submit",
command=self.choose_user_ui, font=("Arial", 18))
            self.submit_btn.pack(pady=10)

            self.reset_button()

        def choose_user_ui(self):
            try:
                age = int(self.age_entry.get())
                if age <= 20:
                    self.simple_calculator(fun_ui=True)
                else:
```

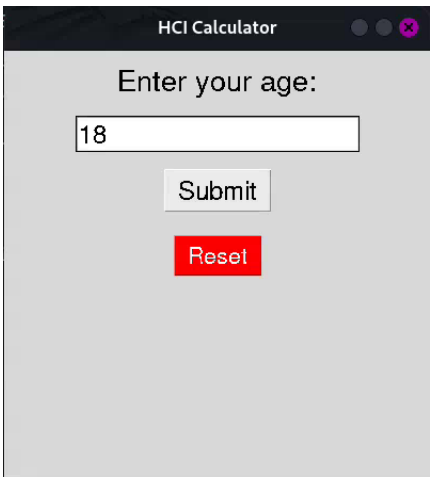
```
                    self.simple_calculator(fun_ui=False)
            except ValueError:
                messagebox.showerror("Invalid Input", "Please enter a valid age.")

        def system_mode(self):
            self.scientific_calculator()

        def simple_calculator(self, fun_ui):
            self.clear_widgets()
            colors = ["lightblue", "lightgreen", "lightpink", "lightyellow"] if
fun_ui else ["white"] * 4
            self.entry = tk.Entry(self.root, width=20, font=("Arial", 18), bd=5)
            self.entry.grid(row=0, column=0, columnspan=4, padx=10, pady=10)

            buttons = [
                ('7', 1, 0), ('8', 1, 1), ('9', 1, 2), ('+', 1, 3),
                ('4', 2, 0), ('5', 2, 1), ('6', 2, 2), ('-', 2, 3),
                ('1', 3, 0), ('2', 3, 1), ('3', 3, 2), ('*', 3, 3),
                ('0', 4, 0), ('.', 4, 1), ('=', 4, 2), ('/', 4, 3),
            ]

            for (text, r, c), color in zip(buttons, colors * 4):
                tk.Button(self.root, text=text, width=5, height=2, font=("Arial",
14), bg=color,
                          command=lambda t=text: self.on_click(t)).grid(row=r,
column=c)

            self.reset_button(row=5, columnspan=4)

        def scientific_calculator(self):
            self.clear_widgets()
            self.entry = tk.Entry(self.root, width=25, font=("Arial", 18), bd=5)
            self.entry.grid(row=0, column=0, columnspan=5, padx=10, pady=10)

            buttons = [
                ('7',1,0), ('8',1,1), ('9',1,2), ('+',1,3), ('log',1,4),
                ('4',2,0), ('5',2,1), ('6',2,2), ('-',2,3), ('exp',2,4),
                ('1',3,0), ('2',3,1), ('3',3,2), ('*',3,3), ('sin',3,4),
                ('0',4,0), ('.',4,1), ('=',4,2), ('/',4,3), ('cos',4,4),
                ('C',5,0), ('(',5,1), (')',5,2), ('sqrt',5,3), ('tan',5,4)
            ]

            for (text, r, c) in buttons:
                tk.Button(self.root, text=text, width=5, height=2, font=("Arial",
12),
                          command=lambda t=text: self.on_click(t)).grid(row=r,
column=c)

            self.reset_button(row=6, columnspan=5)
```

```
        def on_click(self, char):
            if char == '=':
                try:
                    expression = self.entry.get()
                    result = str(eval(expression, {"__builtins__": None},
math.__dict__))
                    self.entry.delete(0, tk.END)
                    self.entry.insert(tk.END, result)
                except Exception as e:
                    messagebox.showerror("Error", f"Invalid Expression\n{e}")
            elif char == 'C':
                self.entry.delete(0, tk.END)
            else:
                self.entry.insert(tk.END, char)

        def reset_button(self, row=None, columnspan=1):
            btn = tk.Button(self.root, text="Reset", bg="red", fg="white",
font=("Arial", 16), command=self.main_menu)
            if row is not None:
                btn.grid(row=row, column=0, columnspan=columnspan, pady=10)
            else:
                btn.pack(pady=10)

        def clear_widgets(self):
            for widget in self.root.winfo_children():
                widget.destroy()

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

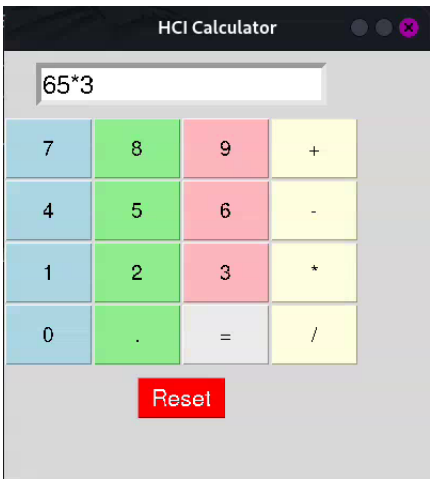
Demo Screens:



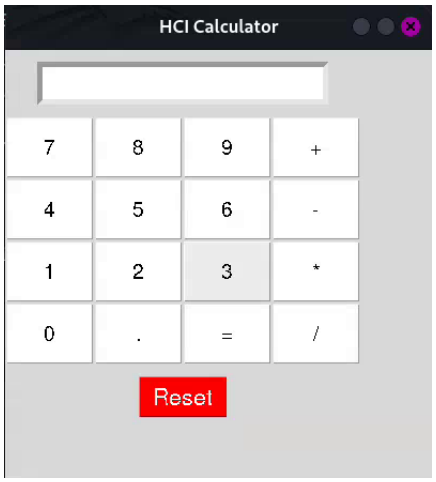
Calculator Mode Selection Screen



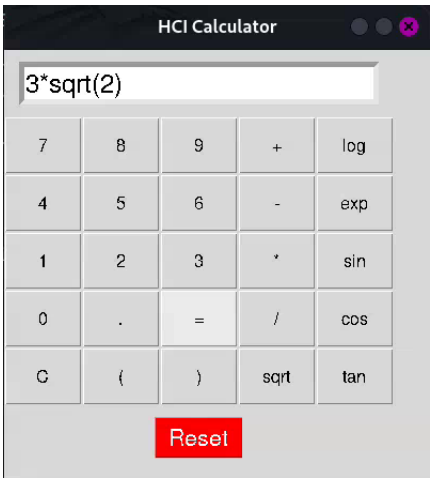
Age input screen



Calculator for Ages-10 to 20



Calculator for Age 20+



System-Centric Calculator

Observation:

- The **User-Centric calculators** adapt based on the user's age, showing that personalization can improve engagement and usability.
- The **System-Centric calculator** focuses on feature completeness, suitable for users who are already familiar with scientific functions.
- The **reset mechanism** enhances the overall user experience by allowing users to easily restart and switch between modes.
- Tooltips provide real-time guidance, improving clarity and decision-making.

Conclusion: This experiment demonstrates the practical differences between user-centric and system-centric designs using **tkinter**. User-Centric design improves the interface for different age groups, focusing on accessibility and user engagement, while System-Centric design emphasizes performance and functional depth. Both approaches serve different purposes and, when combined thoughtfully, can create powerful and inclusive software systems.

Experiment 3

Aim: Implementation of Menus for Graphical System which is to design and implement various types of menus using the Tkinter library to understand menu systems in user interfaces, including Menu Bar, Pull-Down, Cascading, and Pop-up menus.

Theory: A menu in a GUI is a list of options or commands presented to the user. Menus help reduce clutter in the interface and offer easy navigation. Tkinter provides multiple menu types to support a modern GUI system:

1. Menu Bar – A horizontal bar at the top of the window containing drop-down menus like File, Edit, etc.
2. Pull-Down Menu – Menus that drop down when you click on an item in the menu bar.
3. Cascading Menu – Submenus that appear when hovering or clicking on a menu option.
4. Pop-up Menu – Context menus that appear on a right-click anywhere in the application window.
5. Sidebar Menu - Menus that open on the left or right side of the screen by sliding from left to right or right to left.

Tkinter's `Menu` widget is used to create all these menu types. Styling can be achieved using color, fonts, and interaction feedback (e.g., active background/foreground). Event binding like `<Button-3>` is used to trigger pop-up menus. Sidebar menus simulate a mobile navigation drawer using frames and button widgets.

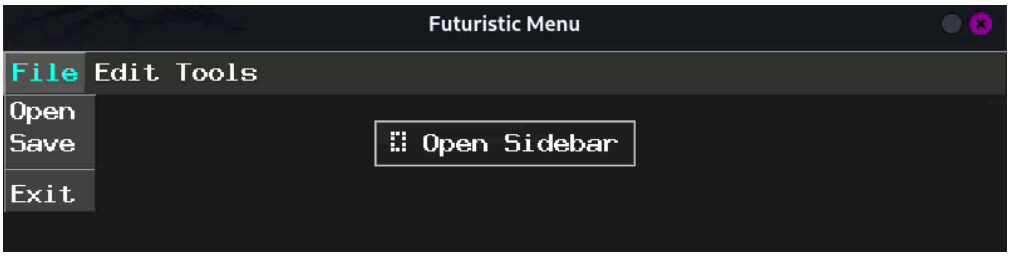
Code:

```
import tkinter as tk
from tkinter import Menu

class StylishMenuDemo:
    def __init__(self, root):
        self.root = root
        self.root.title("Futuristic Menu")
        self.font = ("Roboto", 16, "bold")
        self.root.geometry("800x600+400+50")
        root.resizable(False, False)
        root.configure(bg="#1e1e1e")

        # Menu Bar with Modern Look
        menubar = Menu(root, bg="#333333", fg="white",
activebackground="#555555", activeforeground="cyan")
        root.config(menu=menubar)

        file_menu = Menu(menubar, tearoff=0, bg="#444", fg="white",
activebackground="#555", activeforeground="cyan")
        file_menu.add_command(label="Open", font=self.font)
        file_menu.add_command(label="Save", font=self.font)
        file_menu.add_separator()
        file_menu.add_command(label="Exit", command=root.quit, font=self.font)
        menubar.add_cascade(label="File", menu=file_menu, font=self.font)
```

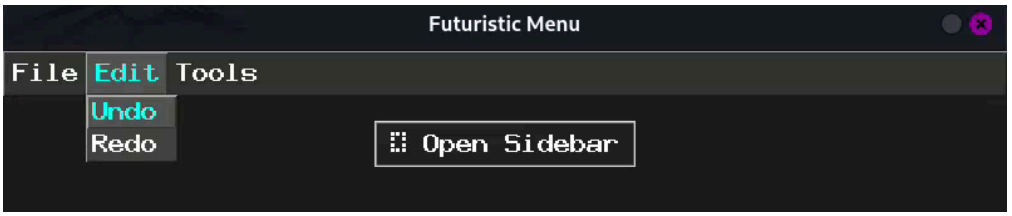
```
        edit_menu = Menu(menubar, tearoff=0, bg="#444", fg="white",
activebackground="#555", activeforeground="cyan")
        edit_menu.add_command(label="Undo", font=self.font)
        edit_menu.add_command(label="Redo", font=self.font)
        menubar.add_cascade(label="Edit", menu=edit_menu, font=self.font)

        tools_menu = Menu(menubar, tearoff=0, bg="#444", fg="white",
activebackground="#555", activeforeground="cyan")
        more_tools_menu = Menu(tools_menu, tearoff=0, bg="#444", fg="white",
activebackground="#555", activeforeground="cyan")
        more_tools_menu.add_command(label="SEO", font=self.font)
        more_tools_menu.add_command(label="UI/UX Design", font=self.font)
        tools_menu.add_cascade(label="More Tools", menu=more_tools_menu,
font=self.font)
        menubar.add_cascade(label="Tools", menu=tools_menu, font=self.font)

        # Pop-up Menu
        self.popup_menu = Menu(root, tearoff=0, bg="#444", fg="white",
activebackground="#555", activeforeground="cyan")
        self.popup_menu.add_command(label="Cut", font=self.font)
        self.popup_menu.add_command(label="Copy", font=self.font)
        self.popup_menu.add_command(label="Paste", font=self.font)
        root.bind("<Button-3>", self.show_popup)

        # Mobile Sidebar Menu
        self.mobile_menu_frame = tk.Frame(root, bg="#333333", width=180,
height=600)
        self.mobile_menu_frame.pack(side=tk.LEFT, fill=tk.Y)
        self.mobile_menu_frame.pack_forget()

        # Buttons inside Sidebar
        self.create_sidebar_buttons()

        tk.Button(root, text="☰ Open Sidebar", command=self.toggle_mobile_menu,
font=self.font,
                  bg="#222", fg="white", activebackground="#555",
activeforeground="cyan", padx=10, pady=5).pack(pady=20)

    def create_sidebar_buttons(self):
        buttons = ["Dashboard", "Settings", "Profile", "Logout"]
        for text in buttons:
            btn = tk.Button(self.mobile_menu_frame, text=text, font=self.font,
bg="#444", fg="white",
                            activebackground="#555", activeforeground="cyan",
padx=10, pady=5)
            btn.pack(fill=tk.X, padx=5, pady=5)

    def show_popup(self, event):
        self.popup_menu.post(event.x_root, event.y_root)
```

```
    def toggle_mobile_menu(self):
        if self.mobile_menu_frame.winfo_ismapped():
            self.mobile_menu_frame.pack_forget()
        else:
            self.mobile_menu_frame.pack(side=tk.LEFT, fill=tk.Y)

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

Demo Screens:

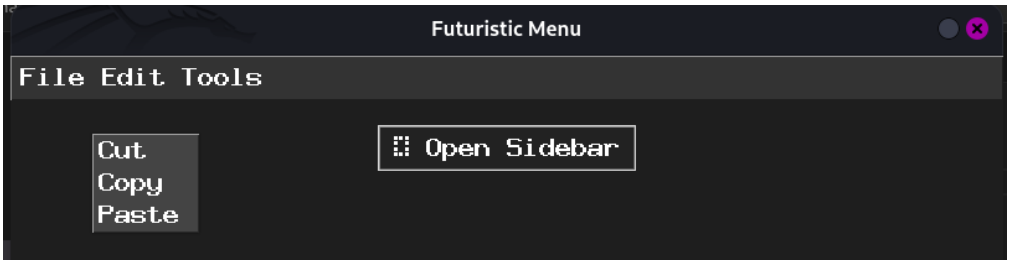


Menu Bar + Drop-Down Menu



Menu Bar + Drop-Down Menu

Drop-Down Menu + Cascading menu



Pop-Up Menu

Sidebar Menu

Observations:

Menu Type	Implemented	Description
Menu Bar	✔️	Contains File, Edit, and Tools menus.
Pull-Down Menu	✔️	"File" and "Edit" menus drop down when clicked.
Cascading Menu	✔️	"More Tools" inside the "Tools" menu is a cascading submenu.
Pop-up Menu	✔️	Right-click shows a context menu with Cut, Copy, and Paste.
Sidebar Menu	✔️	A mobile-style navigation menu appears when "☰ Open Sidebar" is clicked.

All menus are styled with modern themes using dark colors and vibrant highlights to enhance visibility and interaction experience.

Conclusion: The experiment successfully demonstrated various types of menus using the Tkinter library. By implementing a stylish menu system, the interface not only becomes user-friendly but also more engaging and intuitive. This activity helped in understanding the principles of Human-Computer Interaction (HCI) and applying them to create better GUI experiences. Additionally, it encouraged creativity and enhanced programming skills in Python for building interactive applications.

Experiment 4

Aim: Implementation of Different Kinds of Windows like Primary and Secondary Windows which is to design and implement a primary window and various types of secondary windows using Tkinter, demonstrating the flexibility and structure of graphical user interfaces.

Theory: In graphical user interface (GUI) design, windows are the main building blocks for interaction. There are two types:

1. **Primary Window:** The main container window in which the application operates.
2. **Secondary Windows:** Pop-up or supporting windows that assist or interact with the user in specific tasks.

Types of secondary windows implemented:

- **Dialog Boxes:** Temporary modal windows used for short interactions like confirmation or form input.
- **Property Sheets:** Windows with input fields to modify settings.
- **Property Inspectors**: Display editable fields of multiple related properties.
- **Message Boxes**: Predefined pop-up alerts or information messages.
- **Palette Windows**: Floating tool windows, often with buttons or icons.
- **Pop-up Windows**: Small temporary windows that display information or options.

In this experiment, each secondary window is managed such that when the primary window closes, all others are closed too. Iconification behavior (minimize all on minimizing primary) is also handled using events like `<Unmap>` and `WM_DELETE_WINDOW`.

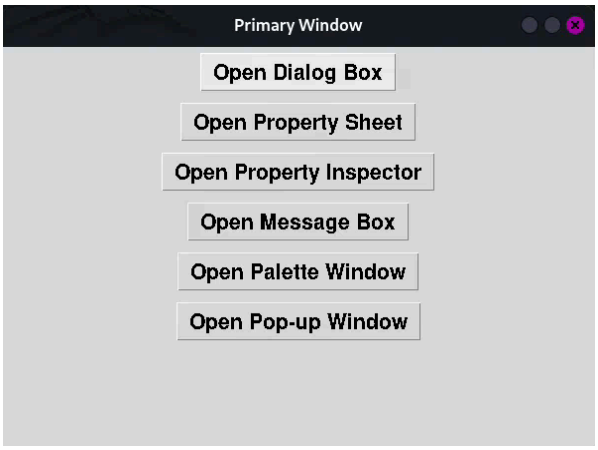
Code:

```
import tkinter as tk
from tkinter import messagebox, Toplevel, PhotoImage

class HCIWindows:
    def __init__(self, root):
        self.root = root
        self.root.title("Primary Window")
        self.root.geometry("600x400")
        self.secondary_windows = []
        self.font = ("Helvetica", 16, "bold")

        # Ensure all secondary windows behave with primary
        self.root.protocol("WM_DELETE_WINDOW", self.on_close)
        self.root.bind("<Unmap>", self.on_minimize)

        # Buttons to open different secondary windows
           tk.Button(root, text="Open Dialog Box", command=self.open_dialog_box,
font=self.font).pack(pady=5)
                              tk.Button(root,    text="Open    Property    Sheet",
command=self.open_property_sheet, font=self.font).pack(pady=5)
                            tk.Button(root,    text="Open    Property    Inspector",
command=self.open_property_inspector, font=self.font).pack(pady=5)
          tk.Button(root, text="Open Message Box", command=self.open_message_box,
font=self.font).pack(pady=5)
```

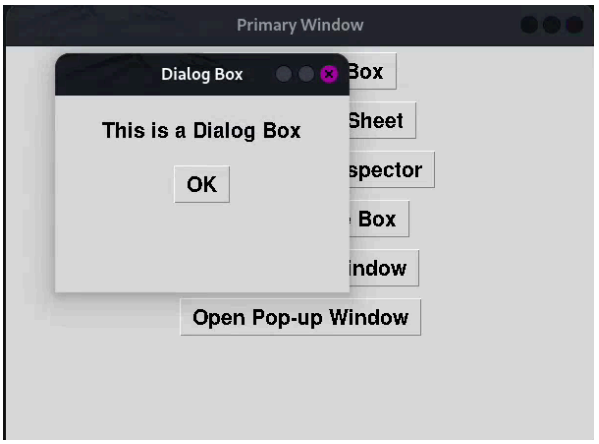
```
                                tk.Button(root,    text="Open    Palette    Window",
command=self.open_palette_window, font=self.font).pack(pady=5)
                                 tk.Button(root,    text="Open    Pop-up    Window",
command=self.open_popup_window, font=self.font).pack(pady=5)

    def open_dialog_box(self):
        window = Toplevel(self.root)
        window.title("Dialog Box")
        window.geometry("300x200")
                          tk.Label(window,   text="This   is   a   Dialog   Box",
font=self.font).pack(pady=20)
                         tk.Button(window,   text="OK",   command=window.destroy,
font=self.font).pack()
        self.track_secondary_window(window)

    def open_property_sheet(self):
        window = Toplevel(self.root)
        window.title("Property Sheet")
        window.geometry("300x250")
                            tk.Label(window,   text="Property   Sheet   Example",
font=self.font).pack(pady=20)
        tk.Entry(window, width=25, font=self.font).pack(pady=5)
                         tk.Button(window,   text="OK",   command=window.destroy,
font=self.font).pack(pady=5)
                       tk.Button(window,   text="Cancel",   command=window.destroy,
font=self.font).pack(pady=5)
        self.track_secondary_window(window)

    def open_property_inspector(self):
        window = Toplevel(self.root)
        window.title("Property Inspector")
        window.geometry("300x300")
        tk.Label(window, text="Property Inspector", font=self.font).pack(pady=10)
        tk.Label(window, text="Property 1: ", font=self.font).pack()
        tk.Entry(window, width=25, font=self.font).pack()
        tk.Label(window, text="Property 2: ", font=self.font).pack()
        tk.Entry(window, width=25, font=self.font).pack()
                        tk.Button(window,   text="Apply",   command=window.destroy,
font=self.font).pack(pady=5)
        self.track_secondary_window(window)

    def open_message_box(self):
        messagebox.showinfo("Message Box", "This is a message box!")

    def open_palette_window(self):
        window = Toplevel(self.root)
        window.title("Palette Window")
        window.geometry("250x200")
        window.configure(bg="gray")
```

```
        icon1 = PhotoImage(file="icons/Default-Icon-icon.png")
        icon2 = PhotoImage(file="icons/Icon-icon.png")

        tk.Label(window, text="Choose an Option:", font=self.font).pack()
                tk.Button(window, text="Option 1", font=self.font, image=icon1,
command=lambda: self.palette_action("Option 1")).pack(pady=5)
                tk.Button(window, text="Option 2", font=self.font, image=icon2,
command=lambda: self.palette_action("Option 2")).pack(pady=5)

        window.icon1 = icon1
        window.icon2 = icon2
        self.track_secondary_window(window)

    def palette_action(self, option):
        messagebox.showinfo("Palette Selection", f"You selected {option}")

    def open_popup_window(self):
        window = Toplevel(self.root)
        window.title("Pop-up Window")
        window.geometry("300x200")
                        tk.Label(window, text="This is a Pop-up Window",
font=self.font).pack(pady=20)
                       tk.Button(window, text="Close", command=window.destroy,
font=self.font).pack()
        self.track_secondary_window(window)

    def track_secondary_window(self, window):
        self.secondary_windows.append(window)
        window.protocol("WM_DELETE_WINDOW", lambda: self.close_window(window))

    def close_window(self, window):
        window.destroy()
        if window in self.secondary_windows:
            self.secondary_windows.remove(window)

    def on_minimize(self, event):
        if self.root.state() == "iconic":
            for win in self.secondary_windows:
                win.iconify()

    def on_close(self):
        for win in self.secondary_windows:
            win.destroy()
        self.root.destroy()

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

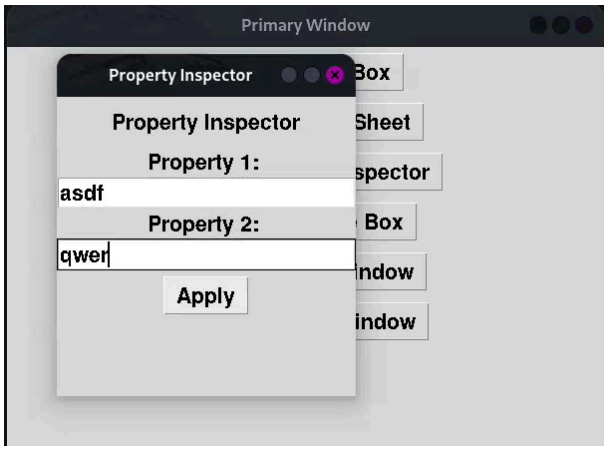
Demo Screens:



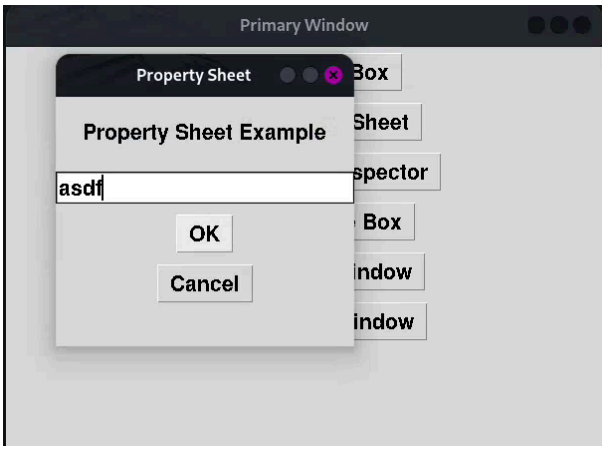
Primary Window



Dialog box window

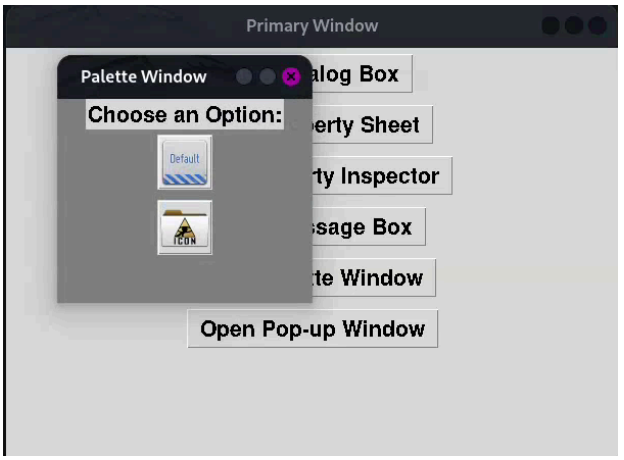


Property Inspector Window

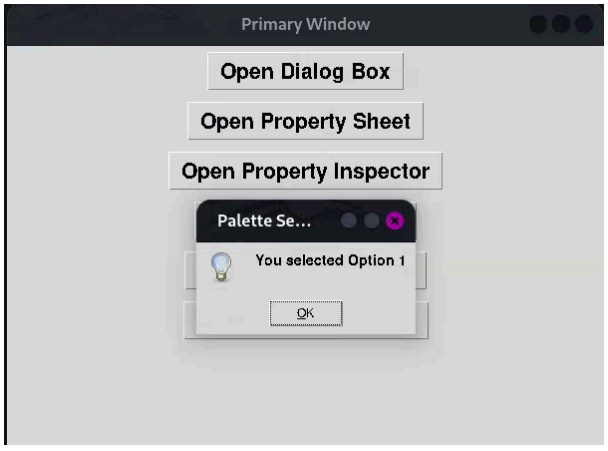


Property Sheet Window

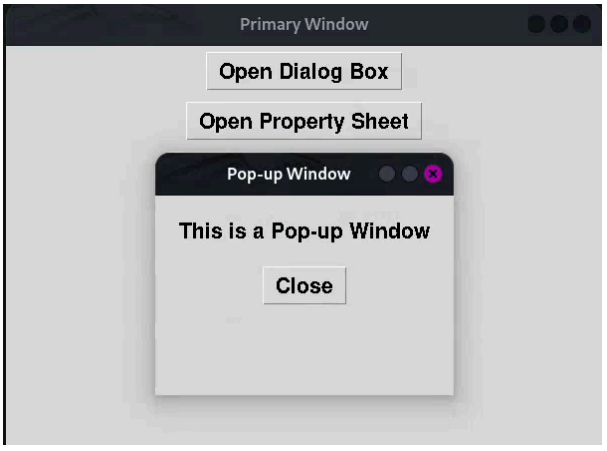
Message Box Window



Palette Window (icon options)



Palette's option 1 pop-up window



Pop-up window

Observation:

Window Type	Implemented	Description
Primary Window	✔️	Main application interface with action buttons.
Dialog Box	✔️	A simple window with a message and OK button.
Property Sheet	✔️	Contains fields and OK/Cancel buttons.
Property Inspector	✔️	Displays and allows editing of multiple properties.
Message Box	✔️	Shows a pop-up info alert.
Palette Window	✔️	Includes options with icons, mimicking a toolbox.
Pop-up Window	✔️	Standalone secondary window with a close button.

All windows use a consistent font and styling. Secondary windows are managed properly on close and minimize actions, enhancing the UI behavior and usability.

Conclusion: This experiment helped in practically understanding how multiple types of windows can coexist and be managed in a GUI system. It also emphasized interaction design principles, especially the coordination between a primary window and its secondary counterparts. Overall, this activity demonstrated key HCI concepts and how they enhance user experience in real-world applications.