# Event-Driven Programming Assignment, Oleksii Nikitin

##### Scenario: You work as a software developer for Future Interactive Technologies Ltd, a small software development company traditionally focused on bespoke database scripts. The company has recently tendered a contract which will involve creating software using the Event Driven Programming (EDP) paradigm. Your manager has asked you to create a sample project demonstrating software developed using the EDP paradigm.

# Task 1 – Planning and Designing an Event-Driven Programming Application (30 Marks)

##### You have been tasked with preparing a software design document for a ‘To-Do List’ graphical user interface (GUI) based application. The application should consider users performing the following operations: Create new entries; View existing entries; Mark an existing entry as complete; and Delete an existing entry. The software design document should at a minimum consider the following; Program Overview, Program Requirements, User Interface Design, Event Handling and Testing Approach.

### Program Overview

The To-Do List application, developed by Future Interactive Technologies Ltd, aims to enhance productivity by enabling users to manage their tasks efficiently. Utilising the Python programming language and Tkinter for the GUI, the application supports operations such as creating, viewing, marking as complete, and deleting tasks. This design document outlines the application's requirements, user interface design, event handling, and testing approach to ensure a comprehensive understanding of its functionality and development considerations.

### Program Requirements

#### Development Environment

The application is developed in Visual Studio Code (VSCode), an extensible code editor that supports Python development on the Python. No dependencies are required.

#### Programming Language

Python 3.12.1 64-bit version has been chosen for its rich ecosystem of libraries and its simplicity in developing GUI applications.

#### Used Libraries:

* **‘tkinter’** will be used to create the Graphical User Interface (GUI) including the use of specific modules and widgets, such as ttk, messagebox, filedialog, Entry, Scrollbar, Button, and Menu.
* **‘os’** is a built-in library to receive relative paths and join paths.
* **‘ctypes’** utilised for Windows-specific functionality, such as setting the application icon on the taskbar (experimental).
* **Cross App imports** facilitate functional relationships between several files.

#### UML/Entity Diagram

A diagram is provided to illustrate the relationship between the main components of the application: **UI**, **Task\_Manager**, **Task**, and **Database\_IO** classes. This helps in understanding the application's architecture.

A diagram of a task

Description automatically generated

#### Coding Style:

* Variable and function names follow the snake\_case naming convention.
* Class names are capitalised and follow the Sentence\_Case convention.
* Function descriptions, comments, and hints are essential conventions in programming that significantly improve the readability of code.

#### Justification:

* **Relative Path:** Importing and receiving the path of the application file to create a relative path to the Database directory.
* **Open File and Re-assign the DB File:** For Application flexibility, we will implement user-defined database storage.
* **Modularization:** It makes any program easier to understand, develop, and maintain. By focusing on smaller parts, we can more easily grasp the functionality of each part, identify, and fix bugs, and enhance or update parts of the program without risking the integrity of the entire system.
* **EDP:** To avoid extra load, we won’t use the built-in Tkinter functionality called after, which launches some checks each timeframe. Instead, we focus on Events and Callback functions that fully facilitate our Application Requirements.
* **Database Saves:** We will implement an Exit question functionality that tracks modification of the task list and saving capabilities through the File Menu to give the user the ability to save files anytime.

### User Interface Design

The design of the To-Do List application's user interface (UI) is centred on creating an intuitive and efficient experience for managing tasks. Tkinter, Python's standard GUI toolkit, was selected for its simplicity, light resource usage, and direct support for event-driven programming, making it an ideal choice for this application. Being a built-in library, it eliminates the need for external dependencies, facilitating easier distribution and setup for end-users. Tkinter also offers straightforward mechanisms for binding events to UI elements, crucial for the responsiveness of an event-driven application like a To-Do List.

#### Alternatives to Tkinter include:

1. **‘Pygame’** isprimarily designed for game development with extensive support for multimedia applications, making it less suited for standard GUI applications due to its complexity and the overheard for simple tasks.
2. **‘PyQt’** offers more features and a modern UI compared to Tkinter. However, it is slightly more complex to use and requires additional setup, making it less ideal for developers seeking simplicity and rapid development.
3. **‘Kivy’** open-source Python library for developing multitouch applications. It is highly versatile but might be more than necessary for straightforward desktop applications, with a learning curve that might not justify its use for simple projects such as the Todo list.

#### Mock-up of Screen

The application features an easily navigable interface with a minimalistic design to avoid overwhelming users. The main window contains a File Menu, an Entry Box bound to the Add button and a Task Treeview list displaying tasks and their status, enabling users to quickly glance at their to-dos. Below the task list, four primary buttons - Complete, Duplicate, Delete, Undo - are positioned strategically for easy access, allowing users to manage their tasks with minimal effort.

The application also facilitates Context Menu on the Todo List Tree, having a unique function as Select All and displaying hotkey shortcuts of those actions.

To visually represent the design, a mock-up will illustrate the application's layout, highlighting the task tree list and the arrangement of control buttons. This mock-up serves as a visual guide to the proposed UI, demonstrating its functionality and user-friendly design ethos.

A screenshot of a computer

Description automatically generated

### Event Handling

Utilising the Event-Driven Programming (EDP) paradigm is pivotal for developing interactive applications like the To-Do List. EDP allows the application to be more responsive and intuitive by executing code in response to user actions or system-generated events, thereby enhancing the user experience.

#### Controls and Events:

* **Entry Box** will be used for creating new tasks. Events include focusing in and out to manage placeholder text and pressing Enter to add a new task.
* **Complete, Duplicate, Delete,** and **Undo Buttons** are click events that trigger bound action for task management. The Add button, for example, is enabled or disabled based on the entry box's content, demonstrating dynamic response to user input. Other buttons are also dynamically disabled depending on the TrueView selection.
* **Task Treeview List** stands for selecting a task, marking it as complete through double-click, or using the context menu to duplicate or delete tasks. These interactions are essential for managing the to-do list.
* **Menu Options** actions like opening a file, saving, or exiting the application are triggered from the file menu, emphasizing a wide range of event-driven functionalities.

#### Event Handlers Implementation:

* **Placeholder Management** for the entry box, event handlers clear or restore placeholder text when the user focuses or defocuses the input field.
* **Task Operations Event** handlers for adding, deleting, marking as complete, and duplicating tasks based on button clicks or keyboard shortcuts.
* **Sorting and Selection** in the task list, event handlers allow sorting tasks by clicking column headers and selecting tasks for operations.
* **Database Interaction** **Events** related to opening, saving, and reloading tasks from a file, ensuring data persistence.

### Approach to Testing

A systematic testing approach will ensure that the To-Do List application functions correctly across different scenarios:

* **Unit Tests** to verify each functionality, such as adding, deleting, or marking tasks as complete, works as intended. This includes testing the underlying logic and event handlers for reliability.
* **GUI Testing** focuses on ensuring that the graphical user interface is responsive and user-friendly. Tests include checking if the UI elements like buttons, entry fields, and the task list behave correctly under various user interactions.
* **Data Validation** ensures the application handles invalid data correctly, including testing for incorrect task inputs and error-handling mechanisms to prevent application crashes.
* **Database tests** the save/load functionality to ensure tasks are correctly persisted across sessions, including edge cases such as saving empty lists or loading corrupted data files.
* **Integrated Testing** combines units to test the application as a whole, ensuring that different components work together seamlessly, such as the interaction between the UI and the backend logic.
* **User Acceptance Testing (UAT)** involves real users testing the application to ensure both functional and non-functional requirements are met. This includes assessing the application's performance, aesthetics, and overall behaviour to ensure it aligns with user expectations.

#### Testing Table Template:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test ID** | **Description** | **Input** | **Expected Results** | **Action** | **Actual Results** |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

This structured approach to testing covers a comprehensive range of test scenarios, from individual functionality to the application's overall performance and user satisfaction, ensuring a reliable and user-friendly To-Do List application.

# Task 2 – Building an Event-Driven Programming Application

##### Using the Python programming language and design document prepared in Task 1, you must now build the ‘To-Do List’ application. Your code must adhere to coding best practices by being consistent, clear and well-documented.

### Description and Screenshots to demonstrate the main features of the application.

#### Title and Dimensions

The application is titled "To-Do List" with a fixed window size of 700x370 pixels, ensuring it doesn't change in size.

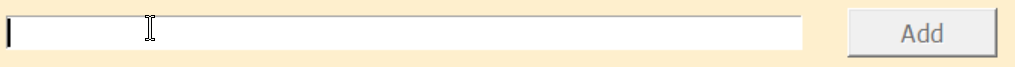
A screenshot of a computer

Description automatically generated

#### Entry Box with Placeholder

A dedicated area allows users to type in their tasks. This box features a placeholder text "Enter your to-do here..." that disappears when the box is focused and reappears if it's left empty. Furthermore, the Add button activates only if there is some text entered.







#### Task List Display

A Tkinter TrueView widget displays tasks with columns for ID, Title, Description, Alarm, and Done status. This widget supports multi-selection, sorting by columns, and features a vertical scrollbar.

A screenshot of a computer

Description automatically generated

#### Control Buttons

Buttons for completing, duplicating, and deleting tasks represent the main functionality. Additionally, dynamically enabled or disabled based on the application's state and user interaction.

A yellow and black text

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

#### Undo Functionality

An undo button suggests the application can revert the last action taken by the user. Additional event handling enables a button if the application history in a session is not empty.

A computer screen shot of a computer error

Description automatically generated A screenshot of a computer

Description automatically generated

#### Menus and Shortcuts

A menu bar at the top includes options like Open, Save, Reload, and Exit, with keyboard shortcuts for convenience. This Menu was created for interaction with CSV database files. We can Open and redefine the CSV storage file. Save to the default one or by your file path. Reload intends to reverse all changes in the list in the application and refresh content from the file if there were manual changes made.

A screenshot of a computer

Description automatically generated

#### Context Menu

Right-clicking on a task opens a context menu with options to complete, duplicate, delete tasks, undo the last action, or select all tasks. Also dynamically enabled depending on related widget statuses: Undo is not active with no history, Complete, Duplicate and Delete actions are not active without selection. Hotkeys are also displayed to demonstrate quick access functionality.

A screenshot of a computer

Description automatically generated A computer screen with a white box

Description automatically generated with medium confidence

#### Event Bindings

The application responds to various keyboard and mouse events, enhancing usability. For instance, pressing **Enter** adds a task, and pressing **Escape** deselects the current widget or selections in the TrueView.

### Complete printout of the code

The Application is split into 3 main files, representing different aspects of functionality that improve navigation through the source code and its readability.

#### **db\_io.py**

import os

from task\_manager import Task

RELATIVE\_PY\_PATH=os.path.dirname(os.path.abspath(\_\_file\_\_)) # Constant to get the current path of the .py file to build relative path

def safe\_cast(value, to\_type, default=None):

    """This function ensures that we won't get an error during the converting.

    Args:

        value (str): String representation of the value that we need to convert.

        to\_type (type): To what type should we convert (int, float, etc)

        default (any, optional): Any value that the user wants to return if the converting failed. Defaults to None.

    Returns:

        type: converted value to required type.    """

    try:

        return to\_type(value)

    except (ValueError, TypeError) as ex:

        print(f"DEBUG: Convertation error - {ex}. Returning default value '{default}'.")

        return default

class Database\_IO:

    """A separate Database class is made to create some private functions that aim to interact with database files."""

    def \_\_init\_\_(self, el\_manager):

        """Database\_IO class initialisation. We receive as an argument the reference to our external Element\_Manager instance.

        Args:

            el\_manager (Element\_Manager): The received reference is to ensure that we use only one list        """

        self.database\_dir:str = RELATIVE\_PY\_PATH + '/database' # relative directory + database work directory

        os.makedirs(self.database\_dir, exist\_ok=True) # Ensure the directory exists, otherwise create it without an error

        self.file\_type='.csv' # just in case we want to change the default extension to txt for example

        self.db\_file\_name:str='db\_todo' # database file name

        self.db\_file\_path:str=os.path.join(self.database\_dir,self.db\_file\_name+self.file\_type) # Building database file path with os.path.join function built-in the os library

        self.el\_manager = el\_manager

    def db\_save\_file(self,file\_path:str=None):

        """This is a public function to save the to-do list to the file

        Args:

            file\_path (str, optional): Defaults (if none) to self.db\_file\_path. The path to our file that we're going to write. The parameter is optional because if we are able to pass the parameter to which file we want to save, we use a predefined database directory, from the relative path that has been built in db\_file\_path        """

        if file\_path is None:

            file\_path = self.db\_file\_path

        if self.\_\_write\_file(file\_path):

            self.is\_modified=False

    def db\_read\_file(self,update\_callback:callable,file\_path:str=None):

        """The public function inside our Database\_IO class is intended to read our database file. We also redefine db\_filepath received from the user in case of usage of the Open File function

        Args:

            update\_callback (callable): As a parameter, we receive a reference to update the interface function and we call it on successful completion of the private \_\_read\_file function

            file\_path (str, optional): Defaults (if none) to self.db\_file\_path. The path to our file. The parameter is optional because if we don't use this function to open the file, we use a predefined database directory, from the relative path that has been built in db\_file\_path        """

        if file\_path is None:

            file\_path = self.db\_file\_path

        if self.\_\_read\_file(file\_path):

            if file\_path != self.db\_file\_path:

                self.db\_file\_path = file\_path # redefine the file path

            update\_callback()

    def \_\_read\_file(self, file\_path: str) -> bool:

        """Private function to read a database file. It skips empty lines, validates 4 data fields in the database CSV file and uses the safe\_cast function to convert the completed status to an integer safely

        Args:

            file\_path (str): Path to the file that we want to read

        Returns:

            bool: True if the file was successfully read and processed, False otherwise.

        """

        try:

            self.el\_manager.todo\_list = []  # Clear the list before refilling it with the new read data

            with open(file\_path, 'r', encoding="utf-8") as csvfile:

                content = csvfile.read()

                lines = content.strip().split('\n')

                for line\_number, line in enumerate(lines, start=1):

                    if not line.strip():  # Skip empty lines

                        continue

                    splitted\_data = line.strip().split(',')

                    if len(splitted\_data) != 4:  # Validate expected data fields

                        print(f"Skipping line {line\_number}: Expected 4 fields, got {len(splitted\_data)}")

                        continue

                    try: # Extract and cast data safely

                        title, description, alarm, completed = splitted\_data

                        completed = safe\_cast(completed, int, 0)  # Ensure priority is an integer, default to 0 if not

                        self.el\_manager.todo\_list.append(Task(title, description, alarm, completed))

                    except ValueError as ve:

                        print(f"Error processing line {line\_number}: {ve}")

                        continue

            return True

        except Exception as ex:

            print(f"Error in reading the file '{file\_path}'\n{ex}")

            return False

    def \_\_write\_file(self, file\_path: str, append: bool = False) -> bool:

        """Private function to write to the database file

        Args:

            file\_path (str): Path to the file that we want to write to.

            append (bool, optional): Defaults to False. If we want to add data to an existing list, this parameter can be set to True to use append mode instead of write mode.

        Returns:

            bool: True if the file was successfully written, False otherwise.

        """

        try:

            if not self.el\_manager.todo\_list: # Check for empty list

                print("Warning: Attempting to write an empty to-do list.")

                return False

            content = ''  # Constructing the content string from the todo list

            for el in self.el\_manager.todo\_list:

                if not hasattr(el, 'title') or not hasattr(el, 'details') or not hasattr(el, 'alarm\_target\_time') or not hasattr(el, 'completed'):

                    print(f"Error: Task object missing required attributes. Skipping task: {el}")

                    continue

                content += f"{el.title},{el.details},{el.alarm\_target\_time},{0 if el.completed == False else 1}\n" #Store True or False in digits

            if not content.strip():  # If content is empty after processing the list

                print("Error: No valid content to write after processing the to-do list.")

                return False

            # Writing the file

            write\_or\_append\_mode = 'a' if append else 'w'

            with open(file\_path, write\_or\_append\_mode, encoding="utf-8") as csvfile:

                csvfile.write(content)

            print(f'Successfully wrote to the file {file\_path}')

            self.el\_manager.is\_modified = False # Reset the modification flag after a successful write

            return True

        except IOError as io\_ex:

            print(f"IOError in writing the file '{file\_path}': {io\_ex}")

            return False

        except Exception as ex:

            print(f"Unexpected error in writing the file '{file\_path}': {ex}")

            return False

#### **task\_manager.py**

class Task:

    def \_\_init\_\_(self,title:str,details:str='',alarm\_target\_time=None,completed:bool=False) -> None:

        """Task class created to represent a task element with certain attributes.

        Args:

            title (str): Task name,

            details (str, optional): Defaults to ''.

            alarm\_target\_time (date/time, optional): Defaults to None. Under development.

            completed (bool, optional): Defaults to False.

        """

        self.title=title

        self.details=details

        self.alarm\_target\_time=alarm\_target\_time

        self.completed=completed

class Task\_Manager:

    """General Task Manager intended to store a list of Tasks, history and flag to identify changes"""

    def \_\_init\_\_(self) -> None:

        self.todo\_list=[]

        self.history = []

        self.is\_modified=False

        # self.add\_dummies() # debug fill

    def add\_dummies(self):

        """Debug function to quickly fill the list"""

        for i in range(1,21):

            self.add\_element(f'Dummy{i}',f'Deets{i}',None,False)

    def add\_element\_by\_index(self,index:int, title:str, details:str='', alarm\_target\_time=None,completed:bool=False):

        """The function adds tasks and receive as parameter index, where it should be placed in the list

        Args:

            index (int): Index of self.todo\_list.

            title (str): Task title.

            details (str, optional): Defaults to ''. Task details.

            alarm\_target\_time (date/time, optional):  Defaults to None. Under development.

            completed (bool, optional): Defaults to False. Store completion of the tasks.        """

        new\_element = Task(title, details, alarm\_target\_time, completed)

        self.todo\_list.insert(index, new\_element)

        self.history.append(('add', index, new\_element)) # Store the action, index, and element object in history

        self.is\_modified=True

    def add\_element(self, title:str, details:str='', alarm\_target\_time=None,completed:bool=False):

        """The function adds tasks to the end of the list.

        Args:

            title (str): Task title.

            details (str, optional): Defaults to ''. Task details

            alarm\_target\_time (date/time, optional):  Defaults to None. Under development.

            completed (bool, optional): Defaults to False. Store completion of the tasks

        """

        new\_element = Task(title, details, alarm\_target\_time, completed)

        self.todo\_list.append(new\_element)

        self.history.append(('add', len(self.todo\_list) - 1, new\_element)) # Store the action, index, and element object in history

        self.is\_modified=True

    def complete\_element(self, indexes):

        """Completes multiple elements, flipping their statutes.

        Args:

            indexes (int): positions to mark tasks done.

        """

        unique\_indexes = set(indexes)

        for index in unique\_indexes:

            if 0 <= index < len(self.todo\_list):

                element = self.todo\_list[index]

                element.completed = not element.completed  # Toggle the completion status

                # element.completed = True  # Sets completion status to True

                self.history.append(('complete', index, element.completed)) # Record this action for undo

        self.is\_modified=True

    def delete\_elements\_by\_ids(self, ids\_to\_delete:list):

        """Delete a range of elements.

        Args:

            ids\_to\_delete (int): Range of indexes.        """

        indexes\_to\_delete\_sorted = sorted(ids\_to\_delete, reverse=True)

        for index in indexes\_to\_delete\_sorted:

            if index < len(self.todo\_list) and index >= 0:

                deleted\_element = self.todo\_list.pop(index)  # pop by index

                # Store the action, index, and element object before deletion

                self.history.append(('delete', index, deleted\_element))

        self.is\_modified=True

    def undo(self):

"""Function to repeat history actions. Just in case we prevent working with no elements in the list"""

        if not self.history:

            return

        action, index, data = self.history.pop()

        if action == 'delete':

            # Insert the element back into its original position

            self.todo\_list.insert(index, data)

        elif action == 'add':

            # Remove the element that was last added

            self.todo\_list.pop(index)

        elif action == 'complete':

            # Revert the completion status

            self.todo\_list[index].completed = not self.todo\_list[index].completed # Switch (Flip) a task status

        self.is\_modified=True

    def set\_alarm(self):

        """Under development"""

        pass

    def edit\_alarm(self):

        """Under development"""

        pass

    def delete\_alarm(self):

        """Under development"""

        pass

#### **gui\_interface.py**

import tkinter as tk

from tkinter import ttk,messagebox,filedialog,Entry,Scrollbar,Button,Menu

import os,ctypes

from task\_manager import Task\_Manager

from db\_io import Database\_IO,RELATIVE\_PY\_PATH,safe\_cast

class UI(tk.Tk):

    def \_\_init\_\_(self) -> None:

        """The initialisation of our GUI interface, as we inherit from tk.Tk parent, we can refer to self as to root. It simplifies interaction with the root. Instead of using self.root. all the time, we refer to self."""

        super().\_\_init\_\_() # Inherits of the root

        self.el\_manager=Task\_Manager() # Initialisation of the instance of Element\_Manager, that will be storing and responsible for interaction with our database list

        self.db\_ref = Database\_IO(self.el\_manager) #  The passage of the reference to ensure that we use only one Element\_Manager.todo\_list

        self.title("To-Do List")

        self.geometry("700x370+700+200")

        self.resizable(width=False, height=False) # Do not allow a user to change the main window size

        self.font=('Tahoma',12) # Default font

        self.entry\_box\_def\_message="Enter your todo here..." # Message for enter\_box Entry, as we use it multiple times, we better store it once

        self.entry\_var = tk.StringVar() # Setup a string variable that will be attached to the entry box to track changes and disable or enable the add button

        if os.name == 'nt': ctypes.windll.shell32.SetCurrentProcessExplicitAppUserModelID() # Apply icon to current app\_id. Usable to apply an icon to Windows' Taskbar using ctypes library. To avoid issues we check if that is Windows to use Windll library. May not work as intended as it's not compiled exe version.

        self.iconbitmap(default=RELATIVE\_PY\_PATH + '/database/ico/to-do-list.ico')

        self.on\_start() # Initialization stage where we create controls and read our db file

        self.protocol("WM\_DELETE\_WINDOW", self.on\_exit\_app) # creating tracks for the even when we close the app with X

        self.last\_sort\_col = None # Sorting variables. Remember the last sorted column.

        self.sort\_reverse = False

        self.dragged\_item = None

        self.scroll\_enabled=False

    # Initialisation

    def on\_start(self):

        """ Initialization stage where we create controls and read our db file."""

        self.create\_controls()

        self.ui\_read\_file()

    def create\_controls(self):

        """The function is responsible to create GUI interface and assign multiple events."""

        # Applying styling to Treeview and main form background

        style = ttk.Style(self) # style.theme\_use("calm")

        style.configure("Treeview", background="#FDE8B4", fieldbackground="#FEEFCD")

        self.configure(background='#FEEFCD')

        # Grid configuration

        self.grid\_columnconfigure(0, weight=1)

        self.grid\_columnconfigure(1, weight=1)

        self.grid\_columnconfigure(2, weight=1)

        self.grid\_columnconfigure(3, weight=1)

        self.columnconfigure(1, weight=3)

        # Entry Box lable used to add elements to the list

        self.entry\_box = Entry(self, font=self.font, fg='grey', textvariable=self.entry\_var)

        self.entry\_box.grid(column=0, row=0, sticky='we', pady=10, padx=(10,140), columnspan=3)

        self.entry\_box.insert(0, self.entry\_box\_def\_message) # Preset default holder message into our enry\_box on creation stage

        self.entry\_box.bind("<FocusIn>", lambda event, m=self.entry\_box\_def\_message, e=self.entry\_box: self.clear\_placeholder(e, m, event))

        self.entry\_box.bind("<FocusOut>", lambda event, m=self.entry\_box\_def\_message, e=self.entry\_box: self.restore\_placeholder(e, m, event))

        self.entry\_box.bind('<Return>', self.add\_task) # Event-driven example on Enter press

        self.entry\_box.bind('<Escape>', self.deselect\_widget) # Deselect the Entry Box on Escape button

        # Setup tracking system for the entry\_box input variable that will call an event each time the value is changing

        self.entry\_var.trace\_add("write", self.on\_entry\_change)

        # Add button creation

        self.add\_btn = tk.Button(self, text='Add', font=self.font, command=self.add\_task, width=10)

        self.add\_btn.grid(column=2, row=0, sticky='e', pady=10, padx=10)

        self.add\_btn.config(state='disabled') # By default, it will be disabled, as no elements selected in a Treeview filed

        self.add\_btn.bind('<Button-1>', self.on\_disabled\_add\_btn) # Even if the button is disabled, we still can track pressing, in our case we use it to focus on the Enry Box

        # Treeview widget used from tkinter.ttk library to display our todo\_list

        self.tree = ttk.Treeview(self, columns=('ID', 'Title', 'Description', 'Alarm', 'Done'), show='headings', selectmode='extended')

        self.tree.heading('ID', text='ID', command=lambda: self.tree\_sort\_column('ID')) # Assigning sorting events

        self.tree.heading('Title', text='Title', command=lambda: self.tree\_sort\_column('Title'))

        self.tree.heading('Description', text='Description', command=lambda: self.tree\_sort\_column('Description'))

        self.tree.heading('Alarm', text='Alarm', command=lambda: self.tree\_sort\_column('Alarm'))

        self.tree.heading('Done', text='Done', command=lambda: self.tree\_sort\_column('Done'))

        # Adjusting column widths for the Treeview

        self.tree.column('ID', width=50, minwidth=50, stretch=tk.NO)

        self.tree.column('Title', width=200)

        self.tree.column('Description', width=250)

        self.tree.column('Alarm', width=100)

        self.tree.column('Done', width=63,stretch=tk.NO)

        self.tree.bind('<<TreeviewSelect>>', self.on\_list\_select) # Event that triggers on selecting any element

        self.tree.bind('<Escape>', self.deselect\_widget) # Escape button to deselect our choices

        self.tree.bind('<Delete>', self.delete\_task) # Escape button to deselect our choices

        self.tree.bind('<Double-1>', self.complete\_task) # Bind double-click event

        self.tree.grid(column=0, row=1, sticky='nsew', pady=10,padx=(10,0), columnspan=3)

        # Adding the scrollbar to our Treeview

        self.scrollbar = Scrollbar(self, orient="vertical", command=self.tree.yview)

        self.tree.configure(yscrollcommand=self.scrollbar.set)

        self.scrollbar.grid(column=3, row=1, sticky='ns', padx=0)

        # Adding Delete button

        self.delete\_btn = Button(self, text='Delete', font=self.font, width=10, command=self.delete\_task)

        self.delete\_btn.grid(column=1, row=2, sticky='e', pady=10,padx=10)

        self.delete\_btn.config(state='disabled') # Disable state by default

        # Adding Done button

        self.done\_btn = Button(self, text='Complete', font=self.font, width=10, command=self.complete\_task)

        self.done\_btn.grid(column=0, row=2, sticky='w', pady=10,padx=10,)

        self.done\_btn.config(state='disabled') # Disable state by default

        # Adding Duplicate button

        self.duplicate\_btn = Button(self, text='Duplicate', font=self.font, width=10, command=self.duplicate\_task)

        self.duplicate\_btn.grid(column=1, row=2, sticky='w', pady=10,padx=10)

        self.duplicate\_btn.config(state='disabled') # Disable state by default

        # Adding Undo button

        self.undo\_btn = Button(self, text='Undo', font=self.font, width=10, command=self.undo)

        self.undo\_btn.grid(column=2, row=2, sticky='e', pady=10, padx=10)

        self.undo\_btn.config(state='disabled') # Disable state by default

        # Setup menu

        self.menu\_bar = Menu(self)

        self.config(menu=self.menu\_bar)

        # File menu

        self.file\_menu = Menu(self.menu\_bar, tearoff=0, bg="#FEEFCD")

        self.menu\_bar.add\_cascade(label="File", menu=self.file\_menu)

        self.file\_menu.add\_command(label="Open", command=self.on\_open\_file, accelerator="Ctrl+O")

        self.file\_menu.add\_command(label="Save", command=self.ui\_save\_file, accelerator="Ctrl+S")

        self.file\_menu.add\_command(label="Reload", command=self.ui\_read\_file, accelerator="Ctrl+R")

        self.file\_menu.add\_separator()

        self.file\_menu.add\_command(label="Exit", command=self.on\_exit\_app, accelerator="Ctrl+Q")

        # Bind hotkeys

        self.bind('<Control-o>', self.on\_open\_file)

        self.bind('<Control-s>', self.ui\_save\_file)

        self.bind('<Control-r>', self.ui\_read\_file)

        self.bind('<Control-q>', self.on\_exit\_app)

        self.bind('<Control-d>', self.duplicate\_task)

        self.bind('<Control-z>', self.undo)

        self.bind('<Control-c>', self.complete\_task)

        self.tree.bind('<Control-a>', self.select\_all)

        # Create a context menu

        self.context\_menu = tk.Menu(self, tearoff=0)

        self.context\_menu.add\_command(label="Complete", command=self.complete\_task, accelerator="Ctrl+C")

        self.context\_menu.add\_command(label="Duplicate", command=self.duplicate\_task, accelerator="Ctrl+D")

        self.context\_menu.add\_command(label="Delete", command=self.delete\_task, accelerator="Delete")

        self.context\_menu.add\_separator()

        self.context\_menu.add\_command(label="Undo", command=self.undo, accelerator="Ctrl+Z")

        self.context\_menu.add\_separator()

        self.context\_menu.add\_command(label="Select All", command=self.select\_all, accelerator="Ctrl+A")

        # Bind the right-click context menu

        self.tree.bind("<Button-3>", self.show\_context\_menu)

    # DB file interaction

    def on\_open\_file(self,\*\_): # Extra function to ensure to save changes if there are some and call

        """This is the event that triggers when we have unsaved changes before, we open a new file.

        If there were any changes, the flag self.el\_manager.is\_modified will be True, in this case we call function on\_unsaved\_changes and pass our ui\_open\_file callback function when we done asking to save changes. Otherwise, we just call ui\_open\_file to open the file."""

        if self.el\_manager.is\_modified:

            self.on\_unsaved\_changes(self.ui\_open\_file)

        else:

            self.ui\_open\_file()

    def ui\_open\_file(self):

        """This UI function asks a user to select a database file, and if we have a new path, we redefine the default path and will be working with that file from now on."""

        file\_path = filedialog.askopenfilename(

            initialdir=self.db\_ref.database\_dir,  # Starting directory for the open\_file dialog

            title="Select file",

            filetypes=(("Text or CSV database file", "\*.txt \*.csv"), ("all files", "\*.\*"))  # File filters

        )

        if file\_path: # If a file was selected (not cancelled)

            if self.db\_ref.db\_read\_file(self.update\_ui\_tree,file\_path): # If we read the file and pass a new file path, also pass our update UI function to update the tree with new information

                self.el\_manager.is\_modified = False # We read fresh new data, and set is\_modified to false to reflect the fresh stage for the on\_unsaved\_changes question

    def ui\_save\_file(self,\*\_):

        """An internal function-event intends to call the external db\_save\_file function from the DB class."""

        self.db\_ref.db\_save\_file()

    def ui\_read\_file(self,\*\_):

        """An internal function-event intends to call the external db\_read\_file function from the DB class."""

        self.db\_ref.db\_read\_file(self.update\_ui\_tree)

    def on\_exit\_app(self,\*\_):

        """A function-event intends to call the on\_unsaved\_changes function-question and as a callback, we pass root.quit() if we finish successfully, otherwise if changes have not been detected, we exit the application."""

        if self.el\_manager.is\_modified:

            self.on\_unsaved\_changes(self.quit)

        else:

            self.quit()

    def on\_unsaved\_changes(self,callback:callable):

        """Function Event if changes have been detected we call save and callback optional function.

        Args:

            callback (callable): This callable variable we use to pass execution in succesfull cases.

        Returns: If user\_choice is None, it means Cancel was selected, so we do nothing and return from the function."""

        user\_choice = messagebox.askyesnocancel("You have unsaved changes", "Do you want to save changes?")

        if user\_choice is True:  # Save

            self.db\_ref.db\_save\_file()

            if callback: callback()

        elif user\_choice is False:  # Don't Save

            if callback: callback()

        return # If user\_choice is None, it means Cancel was selected, so we do nothing and return from the function

    # Actual functionality

    def update\_ui\_tree(self):

        """Updated our Tree in a way to recreate it, it's a more efficient way to interact with the Tree instead of iterating through the tree values to delete some index."""

        self.undo\_button\_state\_check()

        for item in self.tree.get\_children():

            self.tree.delete(item)

        for inx,el in enumerate(self.el\_manager.todo\_list):

            done\_status = '「✔」' if el.completed==1 else '「    」'

            self.tree.insert('', 'end', values=(inx+1, el.title, el.details, el.alarm\_target\_time, done\_status))

    def add\_task(self,\*\_):

        """The function adds task elements to the to-do storage list. Also, we update the tree and clear previous entry and out-focus the entrybox

        This function also has an additional optional parameter to make it callable as an event."""

        title = self.entry\_box.get().strip()

        if title != self.entry\_box\_def\_message and title != "":

            self.el\_manager.add\_element(title)

            self.update\_ui\_tree()

            self.entry\_box.delete(0, tk.END)

            self.focus()

    def delete\_task(self,\*\_):

        """Deletion number of tasks, possible multiple selections and delete multiple. And updates the tree. Has DELETE hotkey.

        This function also has an additional optional parameter to make it callable as an event."""

        selected\_indices = [self.tree.index(item) for item in self.tree.selection()] # Retrieve the correct index

        self.el\_manager.delete\_elements\_by\_ids(selected\_indices) # delete elements by indexes

        self.update\_ui\_tree()

    def complete\_task(self,\*\_):

        """Completing tasks with multiple selections. And updates the tree.

        This function also has an additional optional parameter to make it callable as an event."""

        indexes\_to\_toggle = [self.tree.index(item) for item in self.tree.selection()]  # Retrieve the correct index

        self.el\_manager.complete\_element(indexes\_to\_toggle) # call the complete method with the list of indexes

        self.update\_ui\_tree()

    def undo(self,\*\_):

        """Undo added, deleted or completed tasks one action by the time. And updates the tree. Has CTRL+Z hotkey.

        This function also has an additional optional parameter to make it callable as an event."""

        self.el\_manager.undo()

        self.update\_ui\_tree()

    def undo\_button\_state\_check(self):

        """In each update tree call, we check if there is history for UNDO action, if there is, we enable the button, otherwise history has no stored actions. The same is for context menu option."""

        if len(self.el\_manager.history)>0:

            self.undo\_btn.config(state='normal')

            self.context\_menu.entryconfig("Undo", state="normal")

        else:

            self.undo\_btn.config(state='disabled')

            self.context\_menu.entryconfig("Undo", state="disabled")

    def duplicate\_task(self, \*\_):

        """Duplication of the task, it supports multiple selections and places duplications after each currently selected task (by index in the list).

        This function also has an additional optional parameter to make it callable as an event."""

        selected\_items = self.tree.selection()  # Get the selected item(s)

        for item in reversed(selected\_items):

            item\_values = self.tree.item(item, 'values')

            index\_to\_insert = safe\_cast(item\_values[0], int, 0) # Duplicate at the calculated index

            self.el\_manager.add\_element\_by\_index(index\_to\_insert, \*item\_values[1:-1]) # \*item\_values[1:-1] correctly represents the additional parameters your method expects

        self.update\_ui\_tree()

    def select\_all(self,\*\_):

        """Select All function when the tree is focused. Has CTRL+A hotkey.

        This function also has an additional optional parameter to make it callable as an event."""

        self.tree.selection\_set(self.tree.get\_children())

    # Events

    def show\_context\_menu(self, event):

        """Displays context menu on Right Mouse Button press without altering selection. Operates on already selected items.

        If no items are selected, it disables certain actions."""

        try:

            if self.tree.selection():

                # Enable actions if items are selected

                self.context\_menu.entryconfig("Complete", state="normal")

                self.context\_menu.entryconfig("Duplicate", state="normal")

                self.context\_menu.entryconfig("Delete", state="normal")

            else:

                # Disable actions if no items are selected

                self.context\_menu.entryconfig("Complete", state="disabled")

                self.context\_menu.entryconfig("Duplicate", state="disabled")

                self.context\_menu.entryconfig("Delete", state="disabled")

            # Show the context menu at the cursor's location in any case

            self.context\_menu.post(event.x\_root, event.y\_root)

        except Exception as e:

            print(f'Error in context menu:{e}')

    def tree\_sort\_column(self, col:str):

        """Sorting tree by clicking on a column header. Prioritizing Ascending sorting. Only if you already selected column press the second time - Descending order will apply.

        Args:

            col (str): Column name

        """

        if self.last\_sort\_col != col: # Ensure that we always switch to ascending sorting when we switch to a new column

            self.sort\_reverse = False

        else:

            self.sort\_reverse = not self.sort\_reverse

        if col == 'ID':

            l = [(safe\_cast((self.tree.set(k, col)),int,0), k) for k in self.tree.get\_children('')]

        else:

            l = [(self.tree.set(k, col), k) for k in self.tree.get\_children('')]

        l.sort(reverse=self.sort\_reverse)

        for index, (val, k) in enumerate(l): # Rearrange items in sorted positions (Descending sorting)

            self.tree.move(k, '', index)

        self.last\_sort\_col = col # Reverse sort next time

        self.tree.heading(col, command=lambda: self.tree\_sort\_column(col))

    def on\_entry\_change(self,\*\_):

        """This function-event triggers when entry\_var gets changed to enable/disable add button"""

        content = self.entry\_var.get()

        if content.strip() and content != self.entry\_box\_def\_message:

            self.add\_btn.config(state='normal')

        else:

            self.add\_btn.config(state='disabled')

    def on\_disabled\_add\_btn(self,\*\_):

        """Simple function to focus on the entry box"""

        self.entry\_box.focus()

    def deselect\_widget(self,\*\_):

        """If we have selection inside the tree, we deselect it, otherwise we focus on the root window"""

        if self.tree.selection():

            for item in self.tree.selection():

                self.tree.selection\_remove(item)

        else:

            self.focus()

    def on\_list\_select(self,\*\_):

        """Disable/Enable Done and Delete buttons and their context menus if we have selected at least one task."""

        selected\_items=self.tree.selection()

        if len(selected\_items)<=0:

            self.delete\_btn.config(state='disabled')

            self.done\_btn.config(state='disabled')

            self.duplicate\_btn.config(state='disabled')

        else:

            self.delete\_btn.config(state='normal')

            self.done\_btn.config(state='normal')

            self.duplicate\_btn.config(state='normal')

    def clear\_placeholder(self,entry\_box, default\_message:str,\*\_):

        """Function-event that clear placeholder massage when focused."""

        if entry\_box.get() == default\_message:

            entry\_box.delete(0, tk.END)

            entry\_box.config(fg='black')

    def restore\_placeholder(self,entry\_box, default\_message:str,\*\_):

        """Function-event that restores the placeholder message when unfocused."""

        if entry\_box.get() == "":

            entry\_box.config(fg='grey')

            entry\_box.insert(0, default\_message)

def main():

    app = UI()

    app.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

# Task 3 – Understanding Event-Driven Programming

In addition to the software developed, your manager has requested a report on the following two points to help other developers within the company further understand the EDP paradigm.

## A) Explain the key principles of the Event-Driven Programming paradigm and outline some of the primary ways in which it is used for software development.

Event-driven programming is a paradigm where the flow of the program is determined by events. These events could be anything from user actions (like clicks, and key presses) to system-generated events (like a file being loaded or an error popping up). The core principles of EDP involve event listeners, event handlers (or callbacks), and the event loop.

* **Event Listeners** are parts of the program waiting for events to happen. For instance, a button in a graphical user interface (GUI) listens for clicks.
* **Event Handlers (Callbacks)**. Upon the occurrence of an event, the corresponding event handler is called. This function contains the code that should be executed in response to the event.
* **Event Loop**: This continuously checks for occurring events and dispatches them to their respective handlers. This loop keeps the application running, waiting for user interaction or other events.

We didn’t use Even Loop events like ‘after’ in our application to improve performance, focusing only on Event Listeners and Event Handlers, as the most efficient type of EDP functionality.

In general, EDP is heavily used in applications with a Graphical User Interface (GUI), like To-Do List developed using Tkinter. It allows applications to be responsive, as the event loop is continuously listening for user input without freezing the UI. This model fits well with applications requiring a high degree of interaction from users, such as desktop applications, and web applications, and sometimes for handling asynchronous operations in web servers.

#### Example from To-Do List application developed in Tkinter:

In our application, we've implemented EDP using widgets like buttons. Here's an example with an explanation:

self.add\_btn = tk.Button(self, text='Add', font=self.font, command=self.add\_task, width=10)

self.add\_btn.grid(column=2, row=0, sticky='e', pady=10, padx=10)

self.add\_btn.config(state='disabled')

self.add\_btn.bind('<Button-1>', self.on\_disabled\_add\_btn)

In this snippet, a button is created with the assigned method through ‘command=’, when clicked, calls the add\_task method. This demonstrates how Tkinter facilitates EDP by allowing developers to specify event handlers (add\_task) that respond to events (button click). Even if the button is initially disabled (state='disabled'), binding it to the <Button-1> event with self.on\_disabled\_add\_btn ensures the application can still react to user actions, highlighting the flexibility of EDP.

def add\_task(self,\*\_):

"""The function adds task elements to the to-do storage list. Also, we update the tree, clears previous entries and out-focus the entry\_box.

This function also has an additional optional parameter to make it callable as an event."""

title = self.entry\_box.get().strip()

if title != self.entry\_box\_def\_message and title != "":

self.el\_manager.add\_element(title)

self.update\_ui\_tree()

self.entry\_box.delete(0, tk.END)

self.focus()

This function is an example of an event handler that reacts to the button press event, showcasing how event-driven logic is applied. It processes the input, updates the application state, and refreshes the UI accordingly.

#### Advantages and Disadvantages of EDP:

* One significant **advantage** of EDP is the creation of applications with responsive UIs that can perform background functions. This model allows the UI to remain responsive and interact with the user while waiting for events to occur or processing background tasks, enhancing the user experience.
* A potential **disadvantage** is the complexity and difficulty of testing, as EDP applications can have numerous event sources and handlers, making the flow of the application harder to predict and debug.

#### Common Applications of EDP:

EDP is widely used in various software types, particularly in:

* **GUI Applications**: Desktop applications that require user interaction, such as text editors, design tools, and media players.
* **Web Applications**: Websites and web applications leverage EDP for dynamic content updates and interactive user interfaces without requiring page reloads.
* **Networked Applications**: Server and client applications that respond to network events like incoming data packets.
* **Game Development**: Games rely heavily on EDP for handling user input, game state changes, and rendering graphics based on player actions.

This paradigm's emphasis on waiting for and responding to events as they occur makes it ideal for any application that prioritizes interactivity and responsiveness.

## B) Detail some of the ways in which applications built using EDP can be tested in terms of methodologies, tools and libraries within Python. Provide examples where possible.

Testing applications built with the Event-Driven Programming (EDP) paradigm, especially GUI applications using libraries like Tkinter, poses unique challenges. These challenges stem from the dynamic nature of EDP, where user interactions and system events drive the flow of the application. Unlike linear execution models, EDP applications respond to external stimuli, making their execution paths less predictable and requiring specialized testing approaches.

#### Testing Methodologies and Tools within Python for EDP Applications:

* **Unit Tests:** Focus on testing individual units of code in isolation, such as functions or methods. Python’s built-in unittest library is commonly used for this purpose. Given the event-driven nature of applications, mocking external dependencies and events is crucial. For instance, simulating button clicks or entry inputs to test how the application reacts.
* **Integrated Testing:** Involves testing the interaction between different parts of the application to ensure they work together as expected. This could mean testing the interaction between the user interface and the underlying logic or database. Tools like pytest can be used for more complex test scenarios, including testing asynchronous code, which is common in EDP applications.
* **GUI/User Acceptance Testing:** This tests the application from the user's perspective to ensure it meets the specified requirements. Tools like Selenium with PyAutoGUI for web applications or pytest-qt for Qt applications can simulate user actions (clicks, typing, etc.) and verify the GUI responds as expected. Although Tkinter does not have a dedicated testing library, general-purpose automation tools like PyAutoGUI can be used for simulating user interactions.

#### The Test example of our To-Do List:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test ID** | **Description** | **Input** | **Expected Results** | **Action** | **Actual Results** |
| 1 | **Add Task via Entry Box and Enter press** | Enter "Test Task 1" | The new task appears in the TrueView list | Press 'Enter' | Passed |
| 2 | **Complete Task Action** | Select an existing task | Task status updates to 'Done' | Double-click | Passed |
| 3 | **Delete Task Action** | Select an existing task | The task is removed from the list | Press 'Delete' | Passed |
| 4 | **Undo Last Action** | After deleting a task | Deleted task reappears in the list | Press 'Undo' | Passed |
| 5 | **Open File** | Trigger file open | File content loads into the application | Press 'Open' | Passed |

The screenshots of performed tests: PASSED

1. **Add Task via Entry Box and Enter press:**

A screenshot of a computer

Description automatically generated

1. **Complete Task Action**

A screenshot of a computer

Description automatically generated

1. **Delete Task Action**

A screenshot of a computer

Description automatically generated

1. **Undo Last Action**

A screenshot of a computer

Description automatically generated

1. **Open File**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated