# 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. Utilizing 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.

**Programming Language:** Python, chosen for its rich ecosystem of libraries and its simplicity in developing GUI applications.

**External Libraries:**

* ‘tkinter’ is used for creating the GUI. Specifically, we import ttk, messagebox, filedialog, Entry, Scrollbar, Button, Menu
* ‘os’ is a built-in library to receive relative paths and join paths.
* ‘ctypes’ utilized for Windows-specific functionality, such as setting the application icon on the taskbar (experimental).

**UML/Entity Diagram:** A diagram will be provided to illustrate the relationship between the main components of the application: the UI class, Task\_Manager, and Database\_IO. This will help in understanding the application's architecture.

**Coding Style:**

* Variable and function names follow the snake\_case naming convention.
* Class names are capitalised and follow the Sentence\_Case convention.
* Function description, comments and hints are very important conventions that aim to significantly improve code readability also will be followed.

### 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 Treeview 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: Verify that 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 Interaction: 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.*

asd

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

123