Unit 16 - A2 Object-Oriented Programming

I. Design

Producing designs according to the client requirements.

Requirements

Todo List

This application is meant to demonstrate GUI(Graphical User Interface). Requirements:

- □ Creating, Deleting tasks
- □ Tracking state, and allowing user to change state(Complete, Incomplete)
- □ Support for titles, descriptions, due dates(mutable), completion comments
- □ Displaying tasks in a list
- □ Filtering tasks by state

Index System

This application is meant to demonstrate the use of a database.

Requirements:

- □ The program should be able to read a CSV file
- □ Generate unique index reference for each item
- □ Write the entry to a new CSV file(Without headers)
- □ A separate class responsible for allocation of serial numbers as an interface to allow alternative future implementations

Design

Todo List

a. Problem summary:

The problem is to develop a Todo List application that allows users to create, delete, and manage tasks. The application should support features like tracking the completion status of tasks, setting due dates, and displaying a list of tasks. Users should be able to toggle between displaying all tasks or only incomplete tasks.

b. Complexity:

The complexity of the problem is moderate. It involves managing task data, implementing CRUD (Create, Read, Update, Delete) operations, and providing user-friendly interactions.

c. Constraints:

- Task should have properties like title, description, due date, completion status, and completion comments
- Users should be able to modify task details like title, description, due date, completion status, and completion comments

d. Intended users:

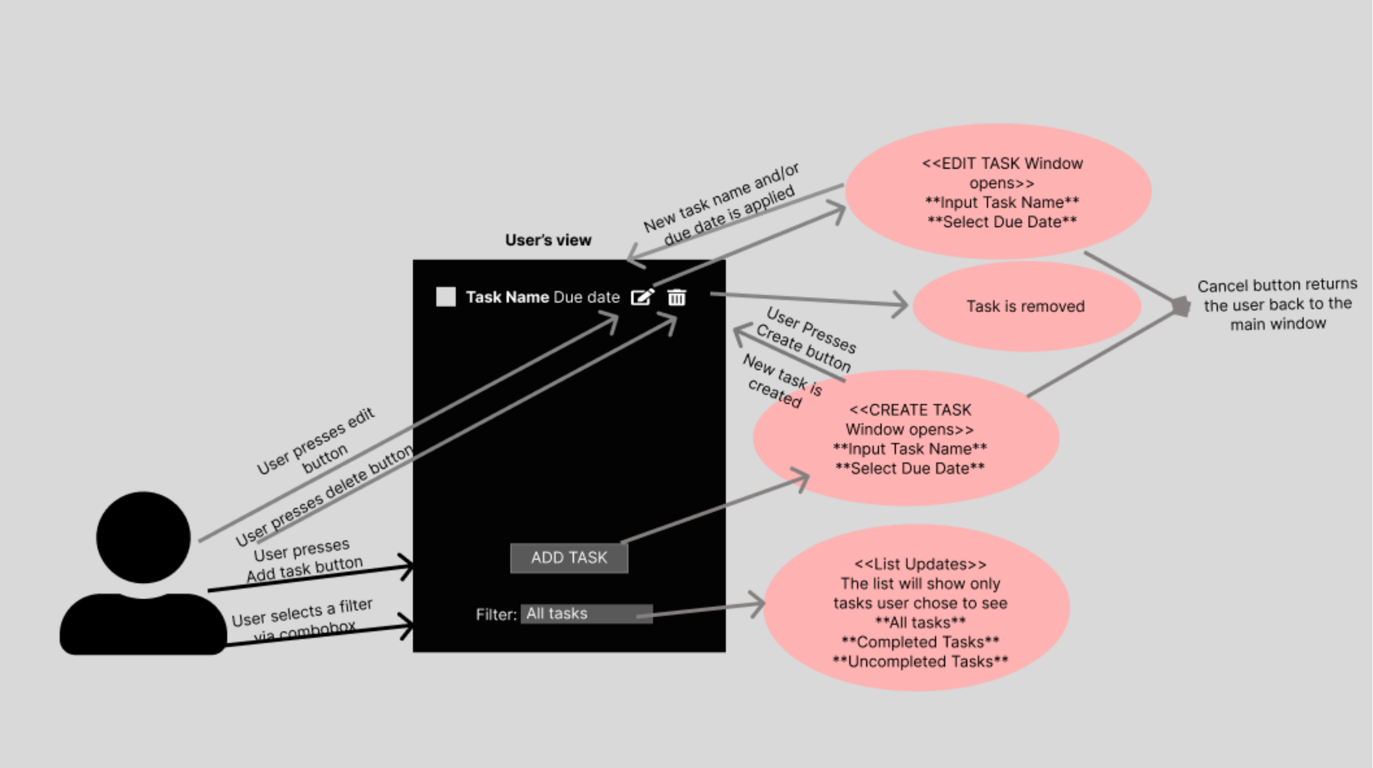
The intended users are people who want to manage their tasks. The application should be easy to use and provide a good user experience.

e. Required Interactivity:

The Todo List application should provide the following interactivity:

- Creating new tasks by entering task details.
- Deleting tasks from the list.
- Tracking the completion status of tasks.
- Modifying task properties such as title, description, due date, and completion status.
- Displaying a list of tasks with filtering options to toggle between all tasks and incomplete tasks.

f. Use case diagram:


User's view
Task Name Due date
ADD TASK
Filter: All tasks
User presses edit button
User presses delete button
User presses Add task button
User selects a filter via combobox
New task name and/or due date is applied
<<EDIT TASK Window opens>> **Input Task Name** **Select Due Date**
Task is removed
User Presses Create button
New task is created
<<CREATE TASK Window opens>> **Input Task Name** **Select Due Date**
Cancel button returns the user back to the main window
<<List Updates>> The list will show only tasks user chose to see **All tasks** **Completed Tasks** **Uncompleted Tasks**

Data Dictionary - Todo List:

1. Data structures:

- Task:
  - title: string
  - description: string
  - due_date: date
  - completion_status: boolean
  - completion_comments: string
- TaskList: A collection (List) of Task objects

2. UI:

- CreateTask: A button which creates an empty task and adds it to the TaskList
- InputTaskDetails: A form which allows users to enter task details(title, description, due date, completion status, and completion comments)
- DeleteTask: A button which deletes the selected task from the TaskList
- ModifyTask: A button which allows users to modify the selected task
- FilterTasks: A button which allows users to toggle between displaying all tasks and displaying only incomplete tasks

3. Data Storage:

- TaskList: A collection (List) of Task objects stored in memory during runtime
- Implementation of persistence storage is out of scope for this project

Index System:

a. Problem summary:

The problem is to develop an Index System application that allows users to generate unique index references for items. The application should support features like reading a CSV file, generating unique index references, and writing the entries to a new CSV file.

b. Complexity:

The complexity of the problem is moderate. It involves reading data from a CSV file, generating unique index references, and writing the indexed data to a new CSV file.

c. Constraints:

- The book details are stored in a CSV file without headings.
- The index references should be unique for each book.

d. Intended users:

The intended users are the staff or administrators of the college library responsible for managing book indexing.

e. Required Interactivity:

The Index System application should provide the following interactivity:

- Reading book details from a CSV file.
- Generating unique index references for each book.
- Writing the indexed data to a new CSV file.

f. Use case diagram:

I do not believe that a use case diagram would be necessary for this application. Since the application is only meant to be opened and run once, there is no need to model the interactions between the user and the application.

Data Dictionary - Index System:

1. Data structures:

- Book:
  - Properties:
    - Name: string
    - Title: string
    - Place published: string
    - Publisher: string
    - Date of publication: date
    - Index reference: string

2. Control structures:

- CSVReader: A class which reads data from a CSV file and returns a list of Book objects

- SerialNumberAllocator: A class which implements an interface to allocate serial numbers

- CSVWriter: A class which writes data to a CSV file

3. Data Storage:

- Input CSV file: A CSV file containing book details(name, title, place published, publisher, date of publication)
- Output CSV file: A CSV file containing book details(name, title, place published, publisher, date of publication, index reference)

4. Pre-defined Code:

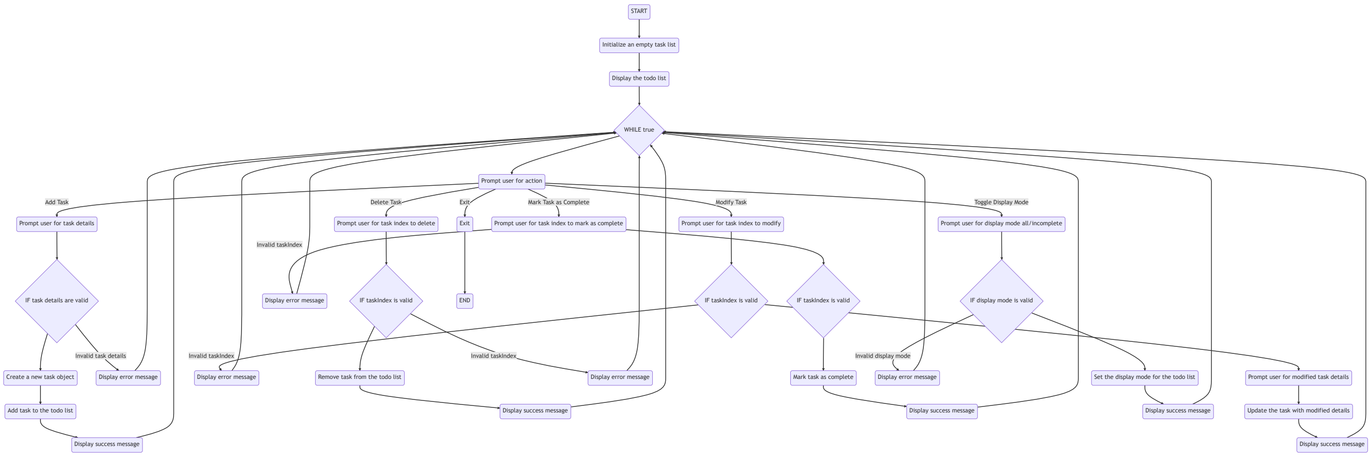
- CSV Parsing Library: Utilize a library or built-in functionality for parsing CSV files and extracting book details from the input file and writing book details to the output file.

Algorithm design - Todo List:

Pseudocode:

```
START

// Initialize an empty task list
todoList = []

// Display the todo list
DISPLAY_TODO_LIST(todoList)

// User interaction loop

WHILE true
    action = PROMPT_USER_FOR_ACTION()

    IF action is "Add Task"
        taskDetails = PROMPT_USER_FOR_TASK_DETAILS()

        IF VALID_TASK_DETAILS(taskDetails)
            task = CREATE_TASK(taskDetails)
            todoList.ADD(task)
        ELSE
            DISPLAY_INVALID_INPUT_MESSAGE()

    ELSE IF action is "Modify Task"
        taskIndex = PROMPT_USER_FOR_TASK_INDEX(todoList)

        IF VALID_TASK_INDEX(taskIndex, todoList)
            modifiedTaskDetails = PROMPT_USER_FOR_MODIFIED_TASK_DETAILS(todoList[taskIndex])

            IF VALID_MODIFIED_TASK_DETAILS(modifiedTaskDetails)
                UPDATE_TASK(todoList[taskIndex], modifiedTaskDetails)
            ELSE
                DISPLAY_INVALID_INPUT_MESSAGE()
        ELSE
            DISPLAY_INVALID_INPUT_MESSAGE()

    ELSE IF action is "Delete Task"
        taskIndex = PROMPT_USER_FOR_TASK_INDEX(todoList)

        IF VALID_TASK_INDEX(taskIndex, todoList)
            todoList.REMOVE(taskIndex)
        ELSE
            DISPLAY_INVALID_INPUT_MESSAGE()

    ELSE IF action is "Mark Task as Complete"
        taskIndex = PROMPT_USER_FOR_TASK_INDEX(todoList)

        IF VALID_TASK_INDEX(taskIndex, todoList)
            todoList[taskIndex].SET_COMPLETED(true)
        ELSE
            DISPLAY_INVALID_INPUT_MESSAGE()

    ELSE IF action is "Toggle Display Mode"
        displayMode = PROMPT_USER_FOR_DISPLAY_MODE()

        IF VALID_DISPLAY_MODE(displayMode)
            todoList.SET_DISPLAY_MODE(displayMode)
        ELSE
            DISPLAY_INVALID_INPUT_MESSAGE()

    ELSE IF action is "Exit"
        BREAK

    DISPLAY_TODO_LIST(todoList)
END
```

Flowchart:



Algorithm design - Index System:

Pseudocode:

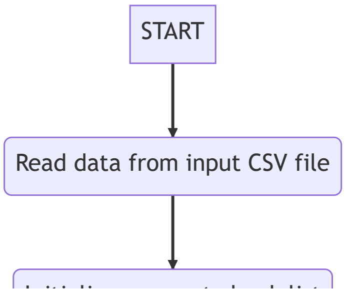
For the main program:

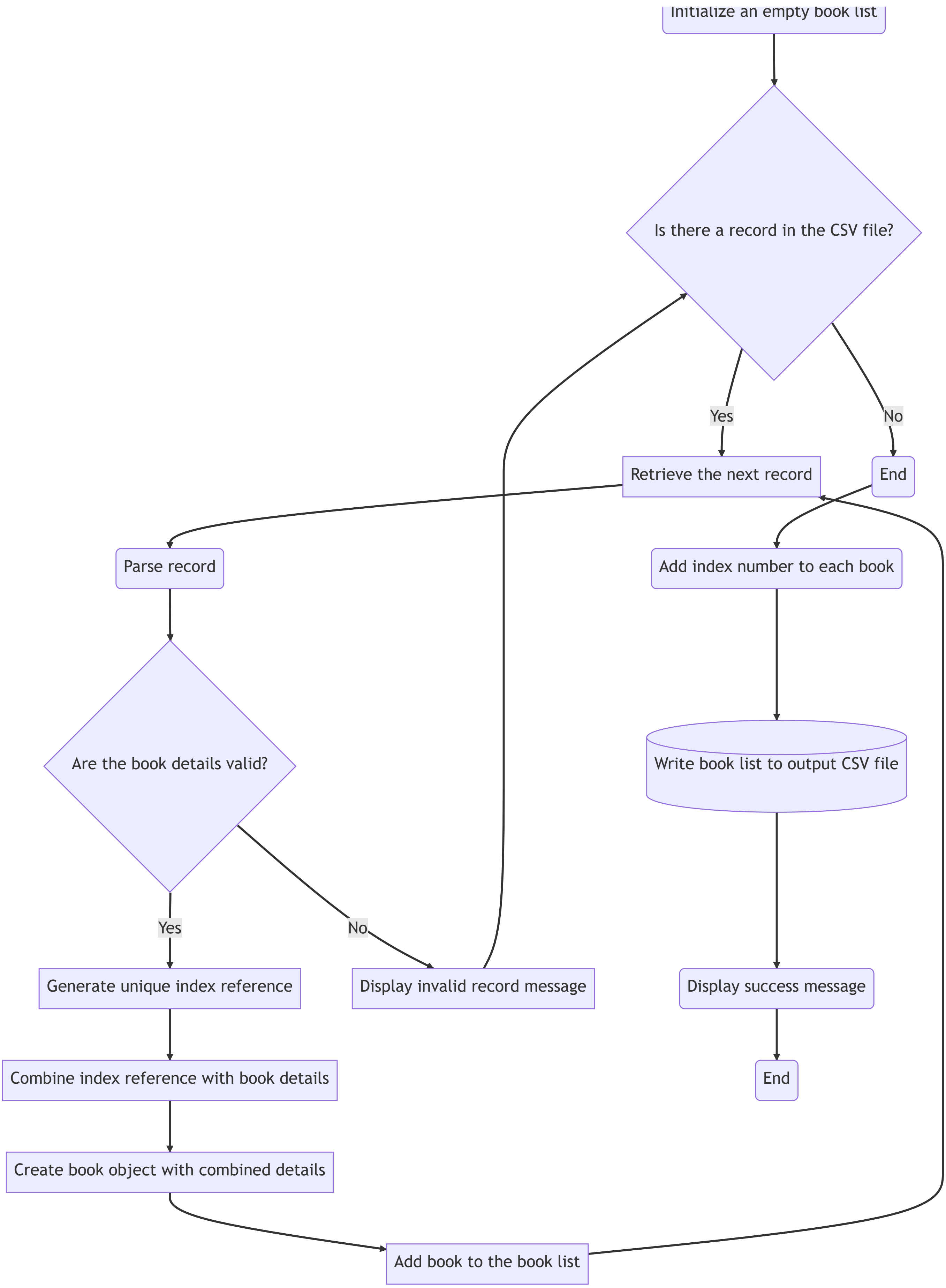
```
START
Read data from input CSV file
Initialize an empty book list

FOR EACH record in the input CSV file
    bookDetails = PARSE_RECORD(record)

    IF VALID_BOOK_DETAILS(bookDetails)
        indexReference = GENERATE_UNIQUE_INDEX_REFERENCE()
        combinedDetails = COMBINE_INDEX_REFERENCE_WITH_BOOK_DETAILS(indexReference, bookDetails)
        book = CREATE_BOOK(combinedDetails)
        bookList.ADD(book)
    ELSE
        DISPLAY_INVALID_RECORD_MESSAGE()

Add index number to each book in the book list

Write book list to output CSV file
Display success message
END
```

For the index reference generator:

```
FUNCTION GENERATE_UNIQUE_INDEX_REFERENCE():
    // Generate a unique index reference using a hashing algorithm
    timestamp = GET_CURRENT_TIMESTAMP()
    randomString = GENERATE_RANDOM_STRING(10) // Generate a random string

    // Combine the timestamp and random string
    dataToHash = CONCATENATE(timestamp, randomString)

    // Apply the hashing algorithm (e.g., SHA-256)
    hashedData = APPLY_HASH_ALGORITHM(dataToHash, "SHA-256")

    // Extract a portion of the hashed data as the index reference
    indexReference = GET_SUBSTRING(hashedData, 0, 8) // Extract the first 8 characters

    RETURN indexReference
```

Flowchart:



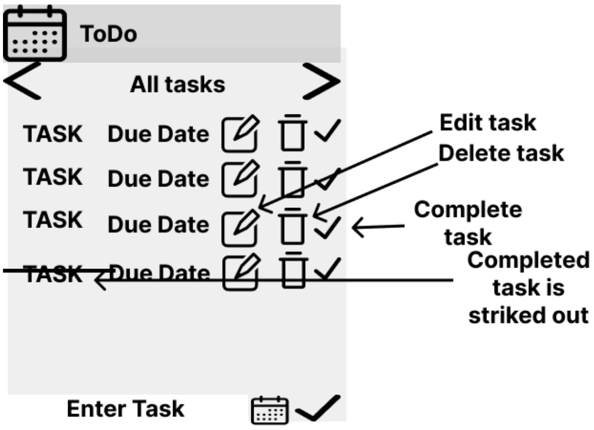

Initialize an empty book list
Is there a record in the CSV file?
Yes
No
Retrieve the next record
End
Parse record
Add index number to each book
Are the book details valid?
Write book list to output CSV file
Yes
No
Generate unique index reference
Display invalid record message
Display success message
Combine index reference with book details
End
Create book object with combined details
Add book to the book list

II. GUI Design

I need to design a GUI for the Todo List application. The GUI should allow users to perform the actions described in the use case diagram. The GUI should also display the current state of the task list.

GUI design - Todo List:

GUI Mock-up:



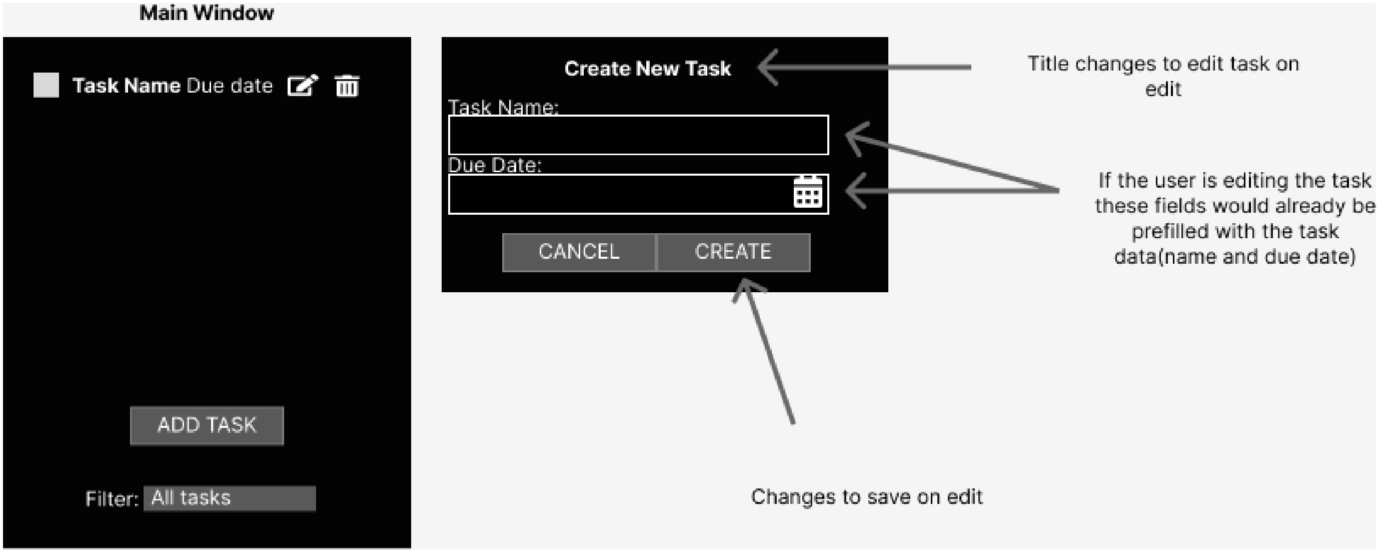
GUI Description:

The GUI consists of the following components:

- A slide filter to filter tasks by their status (All, Active, Completed)
- A list of tasks with the following columns:
    - Task
    - Due Date
    - Edit
    - Delete
    - Mark as Complete Each task will get strike-through text if it is completed.
- A field to enter a new task with buttons to set its due date, and add it to the task list

Name	Feedback
V. Salihu	The GUI is too cluttered. The field for entering a task is too small, and the buttons like data picker and adding the task are too squished together.
S. Coleman	The GUI is not informative. It is not clear what the bottom buttons exactly do. Also missing task completion

Redesign:



Title changes to edit task on edit

If the user is editing the task these fields would already be prefilled with the task data(name and due date)

Changes to save on edit

After careful consideration and feedback, I decided to give the GUI a redesign. The redesign is based on the following considerations:

- I have lacked the skill to implement specific features in the GUI. For example, I have not been able to implement the slider filter, and the strike through text for completed tasks.
- The GUI was too cluttered, the field for entering a task was too small, and the buttons like data picker and adding the task were too squished together.

The redesign fixes some of these issues with the following solutions:

- Slider filter was replaced with a drop-down menu in the bottom. This solution still offers the same functionality as the slider filter, but it is easier to implement.
- Adding a task has become an independent window. This solution allows me to make the field bigger and add more features to the window in the future.
- Completion of the tasks is now done by clicking on the checkbox next to the task. Although this solution is less superior to the strike-through text, I found it easier to implement. Strike-through text is still a feature I would like to implement in the future.
- As a design flaw I did not think about the editing tasks feature. Which I implemented in the redesign.
- I have also added designs of add and edit task windows which are almost identical. This is to make development easier and keep the design consistent across the application.

Name	Feedback
V. Salihu	The redesign is more clean, extra windows means there can be extra features added later without much redesign needed
S. Coleman	The redesign is much better, the GUI is more informative and easier to use, checkbox for completing the task is now added

III. Test Plan

Test Plan - Todo List:

Test Data:

- Creating a task with various characteristics
    - Task Name: Valid task names(“Task 1”, “Task 2”, “Task 3”, “Task 4”, “Task 5”), Empty task name, special characters in task name, task name with length greater than 50 characters.

  - Due Date: Past date, current date, future date, empty date.

- Upon creating a task and verify that the task is added to the list in a correct way.

- Upon creating a task with invalid characteristics, verify that the user is notified of the error and the task is not created.

- Editing a task

  - Correct task is selected for editing
  - The edited task still has the same input constraints as creating a task

- Deleting a task

  - Correct task is deleted

- Filtering tasks

  - All tasks are displayed
  - Only active tasks are displayed
  - Only completed tasks are displayed

- Marking a task as complete

  - Correct task is marked as complete
  - Filtered tasks are updated accordingly

Test Results:

Note: The tests were performed manually, on a Windows 10 machine, using the latest **published** version of the application. This version of the application does not provide any input validation, so only one test case was done just to check that there are no anomalies.

V1.0.1:

ID	Test Case	Test Data	Expected Result	Actual Result	Pass/Fail
1	Creating a valid task	Task Name: “Task 1”, Due Date: 2023-06-04	Task is added to the list	Task is added to the list	Pass
2	Creating a task with no name	Task Name: "", Due Date: 2023-06-04	Task is not added to the list, error message is displayed	The task is added to the list as this version currently has no input validation	Fail

V1.0.2:

Note: The tests were performed manually, on a Windows 10 machine, using the latest **published** version of the application. This version of the application does provide input validation, so all test cases were done.

ID	Test Case	Test Data	Expected Result	Actual Result	Pass/Fail
1.1	Creating a valid task	Name: “Task 1”, Due Date: 2023-06-04	Task is added to the list	Task is added to the list	Pass
2.1	Creating a task with no name	Task Name: "", Due Date: 2023-06-04	Task is not added to the list, error message is displayed	Task is not added to the list, error message is displayed	Pass
3	Creating a task with a name longer than 50 characters	Task Name: "This is a task with a name longer than 50 characters", Due Date: 2023-06-04	Task is not added to the list, error message is displayed	Task is not added to the list, error message is displayed	Pass

ID	Test Case	Test Data	Expected Result	Actual Result	Pass/Fail
4	Creating a task with a past due date	Task Name: "Task 1", Due Date: 2020-06-04	Task is not added to the list, error message is displayed	Task is not added to the list, error message is displayed	Pass
5	Creating a task with a current due date	Task Name: "Task 1", Due Date: 2021-06-04	Task is added to the list	Task is added to the list	Pass
6	Creating a task with a future due date	Task Name: "Task 1", Due Date: 2023-06-04	Task is added to the list	Task is added to the list	Pass
7	Filtering tasks	Filter some preset tasks by their status(All, Uncompleted, Completed)	Tasks are filtered accordingly	Tasks are filtered accordingly	Pass

Note: Editing the task follows the same input validation as creating a task, so only one test case was done.

Test Plan - Index Systems

Note: The tests were performed manually, on a Windows 10 machine.

Test Data:

- Input file is being read correctly
    - Input file should contain the following data:
        - Name
        - Title
        - Place of Publication
        - Publisher
        - Date of Publication
- The records are being assigned an ID
    - The ID is a 6-digit number
    - The ID is unique for each record(Unless the input file contains duplicate records)
- The output file is being created correctly
    - The output file should produce a CSV file with the following columns(No headers)
        - ID
        - Name
        - Title
        - Place of Publication
        - Publisher
        - Date of Publication
- The output file should **NOT** change the order of the records from the input file
- Example input:

Name	Title	Place of Publication	Publisher	Date of Publication
"Orwell, George"	England your England	London	Penguin	2017

- Example output:

805751,"Orwell, George",England your England,London,Penguin,2017

Test Results:

ID	Test Case	Test Data	Expected Result	Actual Result	Pass/Fail
1	Input file is being read correctly	Input file with the following data: Name, Title, Place of Publication, Publisher, Date of Publication	The data is read correctly	The data is read correctly	Pass
2	The records are being assigned an ID	Input file with the following data: Name, Title, Place of Publication, Publisher, Date of Publication	The records are assigned an ID	The records are assigned an ID	Pass
3	The output file is being created correctly	Input file with the following data: Name, Title, Place of Publication, Publisher, Date of Publication	The output file is created with the new 'ID' column and no headers	The output file is created correctly	Pass
4	The output file should **NOT** change the order of the records from the input file	Input file with the following data: Name, Title, Place of Publication, Publisher, Date of Publication	The output file should have the same order as the input file	The output file has the same order as the input file	Pass
5	Test Run	"Orwell, George",England your England,London,Penguin,2017	**Hash Value containing 6-digit number**,"Orwell, George",England your England,London,Penguin,2017	805751,"Orwell, George",England your England,London,Penguin,2017	Pass
6	Behaviour of the application when there is already an output file in the directory	Output file created by the previous run of the application	The application should overwrite the existing output file	The application overwrites the existing output file	Pass
7	Behaviour of the application when there is not output file in the directory	None	The application should create a new output file	The application creates a new output file	Pass

IV. Review of the Implementation

Review of the Implementation - ToDo List:

The solution was implemented using WPF(Windows Presentation Foundation) and C# and the use of Metro UI theme package. It meets most of the requirements in the specification:

- The application allows users to create, edit, delete, and filter tasks.
  - ☑ The application allows users to create and delete tasks.
  - ☑ It tracks the completion of the tasks and allows users to mark tasks as complete.
  - ☑ The application supports titles(mutable) and due dates(mutable) for tasks.
  - ☑ The application allows users to filter tasks by their status(All, Uncompleted, Completed).
  - ☑ The application displays the tasks in a list.
- The application has not met the following requirements:
  - □ The application supports descriptions for tasks.
  - □ The application supports completed date or completed description for tasks.

Reasons for not meeting the requirements:

Due to the time constraints, I was not able to implement all the features I needed to. Because of the time it took to learn WPF, I have put all the attention on optimizing the application and adding features that are essential for the application like: input validation, filtering, and editing tasks. With more time, I would have been able to implement the rest of the features required.

Displaying various Object-Oriented Programming principles:

The application displays the following OOP principles:

- **Encapsulation**: The application uses encapsulation to hide implementation details, for example:
  - The TaskModel.cs class data such as task names and due dates in your TaskModel class are encapsulated within private fields (taskName and dueDate) and accessed through public properties (TaskName and DueDate). This encapsulation provides control over how the data can be accessed and modified.

```
        public class TaskModel : ObservableObject
        {
        private string taskName; // Private field for task name
        private DateTime dueDate; // Private field for due date

        public string TaskName
        {
            get { return taskName; }
            set { SetProperty(ref taskName, value); }
        }

        public DateTime DueDate
        {
            get { return dueDate; }
            set { SetProperty(ref dueDate, value); }
        }

        // Other members of the TaskModel class...

        private void EditTask()
        {
            // Code for editing a task...
        }

        private void DeleteTask()
        {
            // Code for deleting a task...
        }
    }
```

  - Hiding the DeleteTask and EditTask methods in the TaskModel class helps encapsulate the behavior and provides control over how tasks can be modified. By making these methods private and only accessible within the TaskModel class, you limit their visibility and prevent external entities from directly modifying tasks in an undesirable way. This encapsulation serves as a form of access control and ensures that modifications to tasks can only be performed through the designated public methods and properties provided by the TaskModel class.

    ```
            public class TaskModel : ObservableObject
        {
            // Other properties and fields...

            private void EditTask()
            {
                EditTaskWindow editTaskWindow = new EditTaskWindow((MainWindow)App.Current.MainWindow, this).
                editTaskWindow.ShowDialog().
            }

            private void DeleteTask()
            {
                if (Application.Current.MainWindow is MainWindow mainWindow)
                {
                    mainWindow.ListBoxTasks.Remove(this);
                }
            }
        }
    ```

- **Inheritance**: The application uses inheritance to extend the functionality of the MainWindow, CreateTaskWindow and EditTaskWindow classes using the MetroWindow class.

  ```


  #region MainWindow

  public partial class MainWindow : MetroWindow
  {

  }

  #endregion

  #region CreateTaskWindow

  public partial class CreateTaskWindow : MetroWindow
  {

  }

  #endregion

  #region EditTaskWindow

  public partial class EditTaskWindow : MetroWindow
  {

  }

  #endregion
  ```

- **Method Overriding**: The application uses method overriding to override the ToString method in the TaskModel class to return the task name and due date.

```
    public class TaskModel : ObservableObject
    {
        // Other properties and fields...

        public override string ToString()
        {
            return $"{TaskName} (Due: {DueDate:yyyy-MM-dd})";
        }

    }
```

Optimization

Upon receiving feedback and evaluating the initial GUI design, I realized that there were several areas that needed improvement. To address the cluttered interface and lack of clarity in certain features, I decided to redesign the GUI.

The redesigned GUI features a more streamlined and user-friendly interface. The slide filter was replaced with a drop-down menu, simplifying the filtering process. Additionally, I created separate windows for adding and editing tasks, allowing for a more spacious task entry field, and providing room for potential future feature enhancements.

One important feature that was missing in the initial design was task completion. Based on feedback, I added checkboxes next to each task, enabling users to mark tasks as complete. Although I could not implement the strike-through text for completed tasks in this version, it remains a feature I plan to incorporate in the future.

To ensure the effectiveness of the optimizations, I conducted a thorough testing process. The test plan covered various scenarios, including creating tasks with distinctive characteristics, verifying correct task addition and error handling, editing, and deleting tasks, filtering tasks by status, and marking tasks as complete.

During the testing phase, I encountered a discrepancy in the initial version (v1.0.1), where tasks with empty names were erroneously added to the list. However, in the subsequent version (v1.0.2), I implemented input validation, which resolved the issue. The remaining test cases for both versions passed successfully.

Overall, the optimizations made to the ToDo list application improved the user experience by enhancing the GUI's cleanliness, intuitiveness, and functionality. The feedback received from users played a crucial role in driving these improvements, and I am committed to further refining the application based on user needs and preferences.

Review of the Implementation - Index System

The index system was implemented using C# and .NET console application. It meets these requirements in the specification:

- ☑ The program should be able to read a CSV file
- ☑ Generate unique index reference for each item
- ☑ Write the entry to a new CSV file(Without headers)
- ☑ A separate class responsible for allocation of serial numbers as an interface to allow alternative future implementations

The application reads a CSV file and generates a unique index reference for each item. It then writes the entry to a new CSV file without headers. The application uses a separate class responsible for allocation of serial numbers as an interface to allow alternative future implementations.

Displaying various Object Orientated Programming principles:

Inheritance: The code demonstrates inheritance by defining the record BookTwo, which inherits from the base record Book. The BookTwo record adds an additional property called Hash. This inheritance allows BookTwo objects to have all the properties defined in the Book record along with an extra property specific to BookTwo.

```
    public record BookTwo : Book
    {
        public string Hash { get; set; }
    }
```

Abstraction: The code demonstrates abstraction by defining the interface IShortHashGenerator. This interface abstracts the behaviour of generating a short hash by providing a contract (the GetShortHash method) that any implementing class must follow. The ShortHashGenerator class implements this interface to generate a short hash based on an integer hash code.

```
    public interface IShortHashGenerator
    {
        string GetShortHash(int hashCode);
    }
```

Separation of Concerns: The code separates concerns by dividing functionality into different classes. The Program class handles the main program flow, the Utils class encapsulates utility functions, and the Book and BookTwo classes represent data entities. This separation improves code organization and maintainability by isolating various aspects of the system.

Optimization

The index systems application was initially implemented using C# and .NET console application. It had several requirements to fulfil, such as reading a CSV file, generating unique index references for each item, and writing the entries to a new CSV file without headers. To ensure future flexibility, a separate class was implemented to handle the allocation of serial numbers, serving as an interface for alternative implementations.

Throughout the development process, I made use of various Object-Oriented Programming (OOP) principles to improve code organization, maintainability, and extensibility.

To ensure the effectiveness of the optimizations, I developed a comprehensive test plan. The test plan encompassed various scenarios, including verifying correct file reading, assigning unique IDs to records, creating the output file correctly, maintaining the order of records, and ensuring the expected format of the output file. These tests were conducted manually on a Windows 10 machine.

During the testing phase, all test cases produced the expected results, indicating that the application was functioning correctly. Additionally, I incorporated error handling for scenarios such as overwriting an existing output file and creating a new output file when one did not exist.

By implementing these optimizations and conducting thorough testing, the index systems application now performs efficiently, adheres to OOP principles, and meets the specified requirements.