Student Name: *Sean Jordan*

Student No: *20040160*

Lecturer Name: *Kanza Ali Manzar*

Module: *B8IT150 ADVANCED PROGRAMMING (B8IT150_2425_TME4)*

Assignment Title: *Task Tracking System*

No of Words: 900

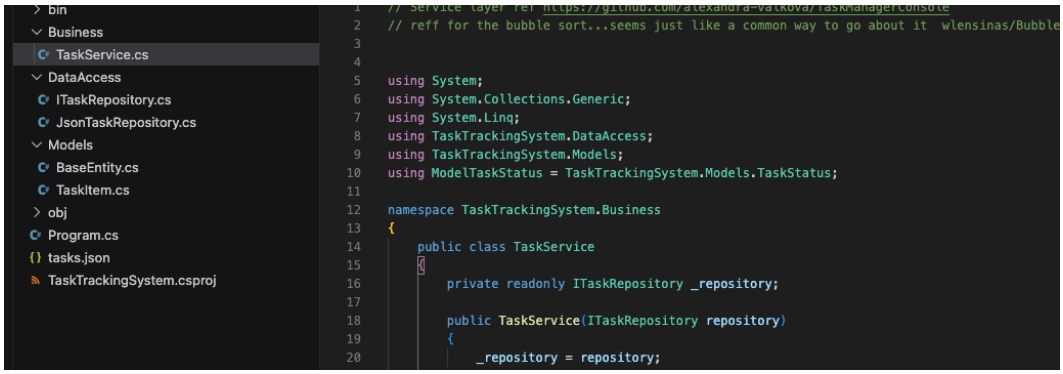
Introduction

For this assignment I built a console-based Task Tracking System using C#. The idea behind it is pretty simple, it just lets a user keep track of tasks through a menu in the terminal. You can add tasks, see them all, search for one by ID, update them, or delete them. All the data is saved to a JSON file so when the program is closed and opened again, the tasks are still there.

I went with a task tracking system because it's easy to understand as a concept, but it still let me cover the main things needed for this module like object oriented programming, working with files, and using algorithms for sorting and searching. I didn't want to overcomplicate it, but I also didn't want everything dumped into one massive file.

How the Program Is Structured

The program isn't all written in one file. Instead, I split it up so different parts of the program do different jobs. I got this idea from looking at small console projects on GitHub and also from examples shown in class. Program.cs is mainly responsible for showing the menu and reading input from the user. It doesn't really contain much logic, it just asks the user what they want to do and then calls the right method. The main logic of the application is in TaskService.cs. This is where tasks are created, updated, sorted and searched. I used this file as a middle layer so the program menu doesn't need to know anything about how tasks are stored. The data itself is handled in JsonTaskRepository.cs, which reads from and writes to a JSON file called tasks.json. This keeps all the file-related code in one place and stops it being mixed in with the rest of the program



Tasks, Data and Storage

Each task is represented by a TaskItem object. A task has an ID, title, description, due date, priority and a status. The status is stored as an enum (Todo, InProgress, Done). I also used a simple base class called BaseEntity that just contains the ID field, so I don't have to repeat it in multiple places.

```
namespace TaskTrackingSystem.Models
{
    public enum TaskStatus
    {
        Todo,
        InProgress,
        Done
    }

    public class TaskItem : BaseEntity
```

Tasks are stored in memory using a List<TaskItem>. When I need to sort or search, I convert this list into an array because it makes writing the algorithms easier.

All tasks are saved to a JSON file using System.Text.Json. When the program starts, it loads the file and turns it into a list of tasks. When anything changes, the file is updated. This approach was based on Microsoft documentation and simple console apps I looked at online. JSON works well here because it's small, readable and doesn't take up much disk space.

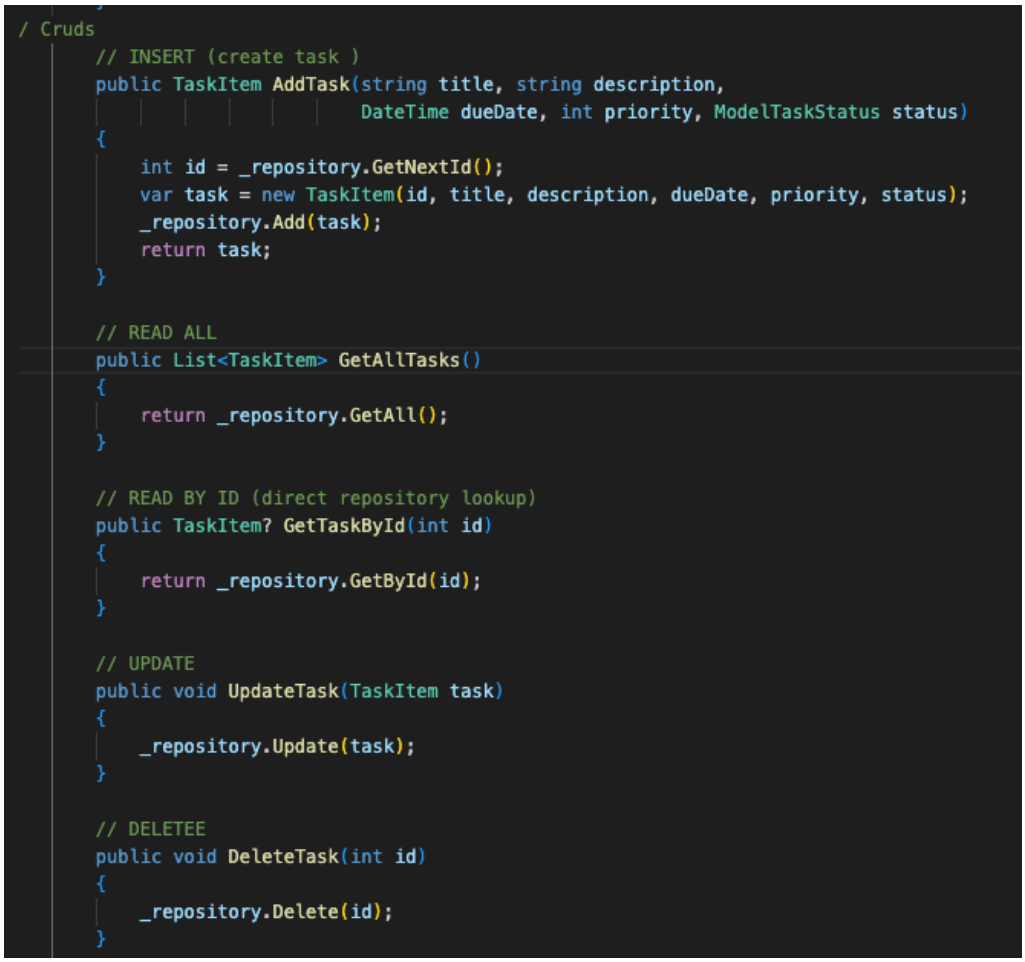
{} tasks.json > ...

```
[
  {
    "Title": "BG_Asset_001",
    "Description": "Locational Asset",
    "DueDate": "2026-01-01T00:00:00",
    "Priority": 2,
    "Status": 0,
    "Id": 2
  },
  {
    "Title": "Silly little task",
    "Description": "Its just a silly fun task",
    "DueDate": "2025-12-31T00:00:00",
    "Priority": 1,
    "Status": 0,
    "Id": 3
  },
  {
    "Title": "Rebirth task that i deleted",
    "Description": "N/A",
    "DueDate": "2030-10-03T00:00:00",
    "Priority": 1,
    "Status": 2,
    "Id": 4
  }
]
```

Main Logic, Sorting and Searching

The TaskService class handles most of what the application actually does. It acts as a middle layer between the menu and the data storage. Instead of letting the menu directly change the JSON file, everything goes through the service class.

Basic things like adding, updating and deleting tasks are done here. It also handles checking whether a task exists before trying to update or delete it, which helps avoid errors.



```
/ Cruds
    // INSERT (create task )
    public TaskItem AddTask(string title, string description,
                            DateTime dueDate, int priority, ModelTaskStatus status)
    {
        int id = _repository.GetNextId();
        var task = new TaskItem(id, title, description, dueDate, priority, status);
        _repository.Add(task);
        return task;
    }

    // READ ALL
    public List<TaskItem> GetAllTasks()
    {
        return _repository.GetAll();
    }

    // READ BY ID (direct repository lookup)
    public TaskItem? GetTaskById(int id)
    {
        return _repository.GetById(id);
    }

    // UPDATE
    public void UpdateTask(TaskItem task)
    {
        _repository.Update(task);
    }

    // DELETEE
    public void DeleteTask(int id)
    {
        _repository.Delete(id);
    }
```

For sorting i used Bubble sort to order tasks by their due date when viewing them, I chose Bubble Sort because it's simple and easy to explain, and the assignment required an actual sorting algorithm. The algorithm compares tasks next to each other and swaps them if they're in the wrong order. It's not super well done but it's fine for a small program like this.

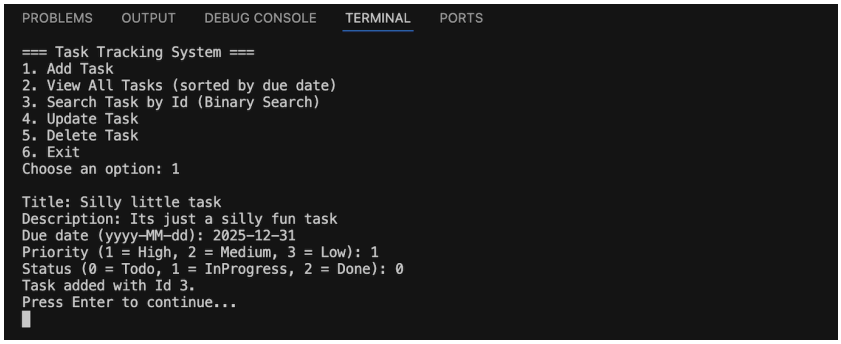
```
//Bubble sorting
// Returns tasks sorted by DueDate using Bubble Sort
public List<TaskItem> GetTasksSortedByDueDate()
{
    var tasks = _repository.GetAll().ToArray();
    BubbleSort(tasks, (a, b) => a.DueDate.CompareTo(b.DueDate));
    return tasks.ToList();
}

// Bubble Sort over TaskItem[] using a Comparison
private void BubbleSort(TaskItem[] array, Comparison<TaskItem> compare)
{
    int n = array.Length;
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (compare(array[j], array[j + 1]) > 0)
            {
                var temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}
```

For searching, I used Binary Search to find a task by its ID. Before searching, the tasks are sorted by ID. The binary search logic follows the standard approach of checking the middle element and narrowing the search range. This shows a more efficient search method compared to just looping through everything, and it meets the assignment requirements.

```
public TaskItem? BinarySearchById(int id)
{
    // Get tasks and sort them by Id first (also using Buble Sort)
    var tasks = _repository.GetAll().ToArray();
    BubbleSort(tasks, (a, b) => a.Id.CompareTo(b.Id));

    int left = 0;
    int right = tasks.Length - 1;

    while (left <= right)
    {
        int mid = (left + right) / 2;
        if (tasks[mid].Id == id)
            return tasks[mid];

        if (tasks[mid].Id < id)
            left = mid + 1;
        else
            right = mid - 1;
    }

    return null;
}
}
```

UI

The user interacts with the program through an simple text menu. The menu allows the user to add tasks, view all tasks, search by ID, update tasks, delete tasks or exit the program. The menu style was inspired by other console-based task manager programs, but adjusted to fit this project. The interface isn’t fancy, I didn't have the mental bandwidth to make something complex, its just plain text in VS terminal.




```c#
static void Main(string[] args)
{
    string filePath = "tasks.json";
    var repository = new JsonTaskRepository(filePath);
    var service = new TaskService(repository);

    while (true)
    {
        Console.Clear();
        Console.WriteLine("=== Task Tracking System ===");
        Console.WriteLine("1. Add Task");
        Console.WriteLine("2. View All Tasks (sorted by due date)");
        Console.WriteLine("3. Search Tas by Id (Binary Search)");
        Console.WriteLine("4. Update Task");
        Console.WriteLine("5. Delete Task");
        Console.WriteLine("6. Exit");
        Console.Write("Choose : ");

        string? input = Console.ReadLine();
        Console.WriteLine();

        switch (input)
        {
            case "1":
                AddTask(service);
                break;
            case "2":
                ViewTasks(service);
                break;
            case "3":
                SearchTask(service);
                break;
            case "4":
                UpdateTask(service);
                break;
            case "5":
                DeleteTask(service);
                break;
            case "6":
                return;
            default:
                Console.WriteLine("Invalid option, Press Entr to continue....");
```

Issues and Final Thoughts

One issue I ran into was a naming conflict with TaskStatus, because .NET already has a built-in type with the same name. This caused compiler errors at first, but I fixed it by being more specific with namespaces. It was a bit annoying but also a good learning experience.

Overall, the Task Tracking System meets the requirements of the assignment. It uses object-oriented design, JSON file storage, data structures and algorithms in a way that's easy to understand and explain. If I had more time, I'd probably add more validation, extra features or maybe perhaps a graphical interface but for the scope of this module project and the time ive had on my hands it works well enough.

References:

Abhi-AIX (2021) task_manager_c_sharp [Online].
https://github.com/Abhi-AIX/task_manager_c_sharp

Ali, K. (2023) Advancd Programming Module Repository [Online]
https://github.com/KAAli-DBS/Advanced-Programming

Microsoft (2023) System.Text.Json serialization in .NET [Online]
https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/how-to

Valkova, A. (2021) TaskManagerConsole [Online]
https://github.com/alexandra-valkova/TaskManagerConsole

Wlensinas (2020) Bubble Sort With C# [Online]
https://github.com/wlensinas/Bubble-Sort-With-C-Sharp