DATING MANAGEMENT SYSTEM

By

Haron- Developer

Abdullahi- Developer

Sultan-Tester

Mehul- Team Leader

Alsir – developer

Joe – tester

Contents

Part I *- Introduction*

Overview of the Project

Report Layout

Part II- *Design*

Data Structure Selection and Justification

Algorithm Analysis

Part III*- Testing*

Testing Approach

Test Cases

Part IV*- Conclusion*

Summary of Work Done

Limitations and Reflections

Future Improvements

Contributions to Project –

- **Report done by Abdullahi & Haron**
- **Testing done by Abdullahi**
- **Program code done by Haron**
- **Design and planning of data structure & algorithms done by Abdullahi and Haron**
- **Readme file done by Haron**
- **Video presentation done by Abdullahi and Haron**
- **Contributions by Sultan,Mehul,Alsir,Joe, Sworup**

Introduction

Our dating management project is a software program designed to enhance and simplify the processes of relationship management and matchmaking. This project is written in C#, which uses SQL to manage data effectively, allowing for secure storage of user information and profiles. The program is designed to perform essential operations like creating, reading, updating, and deleting profiles (CRUD), ensuring reliability and ease of use.

The user-friendly program allows users to manage profiles, keep track of preferences, and build valuable connections. It emphasises accurate data management and scalability, ensuring everything runs smoothly with real-time updates. By integrating C# programming with SQL for data handling, we have developed a practical and user-friendly system.

Through the development of this project, we have applied and demonstrated key software development principles such as object-oriented programming, SQL query writing, and practical application design. This coursework showcases our ability to implement a functional solution while adhering to good development practices.

This report documents the project's development process and outcomes. It explains the choices in designing algorithms and data structures and analyses key functionalities through pseudocode. The testing section details the approach to ensure the system works as intended, backed by specific test cases. Finally, the conclusion reviews the work done, highlights limitations, and presents recommendations for future improvements.

Design

List

Usage: For our project, we used a List inside the ProfileManager class to store profiles. As profiles were added, updated, or removed throughout the program's execution, we required a data structure that would efficiently handle these tasks.

Why We Used It:

- **Dynamic Resizing:** As profiles are added or removed, the list can dynamically resize, allowing us to manage a variable number of profiles. This guaranteed that our system was capable of efficiently managing different datasets.

- **Simple Iteration:** The list structure makes it straightforward to work through profiles, simplifying everyday tasks like filtering, sorting, and displaying data.
- **Easy Filtering and Searching**: LINQ methods such as Where and Any enabled quick and intuitive filtering of profiles based on attributes such as Name, Age, City, or Gender.
- **Sequential Data Access:** User profiles are extracted from the database and stored in memory as a collection through SQL, allowing efficient sequential data access.
- **Compatibility with Database Loading**: Data retrieved from the database via SQL is easily organized into a list, streamlining integration and eliminating the need for additional processing.

Why We Did Not Use Other Structures:

- **Dictionary:**
  We did not use a dictionary because it is not suitable for coding tasks such as sorting, displaying all profiles, or filtering based on several details.
- **HashSet:**
  We did not use HashSet because it does not maintain order.
- **Array:**
  We did not use an array because arrays have fixed sizes and are much less flexible.

By utilising a list, we ensured that our system remained adaptable and straightforward to work with, while reliably handling the dynamic operations required by our application. This approach supported the core functionality of managing profiles in a scalable and maintainable way.

PersonProfile (Custom Class)

Usage: We created the PersonProfile class to represent individual user profiles for our project. This class combines attributes such as Name, Age, Gender, Interests, City, and Country into a single structure. It is the most fundamental process for handling all profile-related data within the program.

Why We Used It:

- Organisation: Arranging all profile information into a single class allowed us to manage data easily and kept everything well structured.
- Reusability: The PersonProfile class, applied across various project parts wherever profile information was needed, ensured consistency and eliminated the need to repeat the same development tasks.
- Clarity: Representing profiles as objects with clearly labelled attributes made navigating the code quicker, easier to understand, and more maintainable.
- Extendibility: This structure allowed the ability to introduce new features to the class without disrupting other program parts.
- Object-Oriented Principles: OOP principles allowed us to create a program that was easy to break down into smaller parts, it was also dependable in operation, and simple to change or fix when required.

Why We Did Not Use Other Structures:

- We avoided dynamic objects as they do not enforce strict type-checking, which could have presented errors during runtime.
- We avoided arrays because they do not support named attributes, making the code harder to understand and work on.
- We avoided dictionaries because, while suitable for mapping data, they lack the structural organisation and clarity of a custom class, which was essential for effectively managing user profiles.

By utilising the PersonProfile class, we ensured a structured, scalable, and reliable way to handle profile-related data, keeping the system consistent and easy to build upon.

Algorithms:

Linear Search Algorithm -

Where It Was Used: Our program used linear search to pinpoint profiles based on specific characteristics. This allowed users to find profiles efficiently by name, age, or city.

What Problem It Solved:
Linear search addressed the need for filtering and searching profiles within an unsorted dataset. The algorithm was straightforward and suited to the nature of our project.

Why We Used It:

- Unsorted Dataset: Profiles were stored in a list that was not pre-sorted, making linear search a practical choice.
- Dynamic Criteria: The algorithm evaluated user input conditions across multiple fields (e.g., name, city, age) without the need for preprocessing or maintaining extra data structures.
- Ease of Implementation: Linear search has been simpler to code, it was easier to maintain, and integrate into our application, reducing development time and the potential for errors.
- Scalability for Small Datasets: With a time, complexity of O(n), linear search is ideal for our project's dataset size, where the number of profiles is relatively small, and performance remains acceptable.

Why We Did Not Use Other Algorithms:

- Binary Search: Binary search requires data to be sorted, which would have added unnecessary complexity to our application.
- Hash-Based Searching: Hash-based searches rely on dictionaries or hash maps and are effective for key-based lookups. However, they are impractical when searching across multiple fields or using dynamic filtering criteria.

Pseudo Code:
Algorithm LinearSearchProfilesByName
Input: profiles (List of PersonProfile), searchName (String)
Output: matchingProfiles (List of PersonProfile)

1. Initialise an empty list matchingProfiles

2. For each profile in profiles:
a. If profile.Name contains searchName (case-insensitive):
Add profile to matchingProfiles
3. Return matchingProfiles

Detailed Breakdown:

- Initialisation: Start with an empty list named matchingProfiles and loop through the profiles list.
- Condition Check: For each profile, check whether the Name field contains the searchName string. The check is case-insensitive, allowing for flexible and forgiving matches.
- Result Aggregation: If a profile meets the condition, add it to the matchingProfiles list.
- Return Results: After processing every profile, return the matchingProfiles list containing all the profiles that matched the search criteria.

Time Complexity:

- Worst Case: If no profiles match the search criteria, the algorithm must check every profile in the list, resulting in an O(n) time complexity.
- Best Case: If the first profile matches (and, where applicable, if the search is designed to terminate early), the effective time may be better. However, in typical filtering scenarios, the algorithm processes the entire list to compile the complete result set.

Regex-Based Matching:

Where It Was Used: Regex was used in Single Detail Search and Multi-Criteria Search for fields like Interests and MatchPreferences.

What Problem It Solved: It enabled flexible and precise searches by allowing case-insensitive and whole-word matching. This simplified complex search requirements and ensured accurate results without manual string handling.

Why We Used It:

- **Dynamic Pattern Matching**: Regex provided a way to precisely match user-input terms in text fields.
- **Case Insensitivity:** The inclusion of RegexOptions.IgnoreCase ensures that the search ignores capitalisation, making it more user-friendly for example a search term like "female" will match "Female," "FEMALE," or "FemaLE."
- **Whole-Word Matching:** Word boundaries (\b) ensured terms matched complete words rather than partial strings.
- **Learning Opportunity**: Integrating Regex offered a chance to explore advanced string-processing techniques for practical use.

Pseudo Code:

Algorithm RegexBasedMatching
Input: profiles (List of PersonProfile), searchTerm (String)
Output: matchingProfiles (List of PersonProfile)

1. Initialise empty list matchingProfiles
2. For each profile in profiles:
    a. If Regex.IsMatch(profile.Interests or profile.MatchPreferences, searchTerm, case-insensitive): Add profile to matchingProfiles
3. Return matchingProfiles

Detailed Breakdown:

- **Initialisation:** Start with an empty list matchingProfiles to store results.
- **Regex Matching:** For each profile, use Regex to check if the Interests or MatchPreferences field matches the user-provided search term. The Regex includes:
- Word Boundaries (\b): To ensure whole-word matching.
- **Escaped Input:** Using Regex.Escape(term) to safely process special characters in the term.
- **Case Insensitivity:** To match user input regardless of capitalization.
- **Result Aggregation:** If a match is found, add the profile to matchingProfiles.
- Return Results: After completing the loop, return the matchingProfiles list.

Time Complexity:

- Worst Case: The Regex checks every profile and evaluates each character in the text fields, resulting in O(n) for the number of profiles multiplied by the length of the text field.
- Best Case: A match is found early in the list, reducing iterations.

Testing:

Statement of Testing Approach

The testing approach for this project was designed to ensure the functionality, reliability, and accuracy of the profile management system, including its integration with the SQL database and its core operations (Create, Read, Update, Delete). The following methods were applied:

Functional Testing:

- Verified each function individually (e.g., profile creation, modification, deletion, and sorting) to ensure they operate as intended.
- Focused on input validation, error handling, and correct outputs for valid and invalid inputs.

Integration Testing:

- Tested the program logic and the SQL database interaction to confirm that profiles were correctly saved, retrieved, updated, and deleted.

Table of Test cases

Functionality Testing:

Test case ID	Functionality	input	Expected output	Actual output	result
19	Profile creation	Name:"Abz", Age: 22, Gender:"male"	Profile added successfully	Profile added successfully	Pass

46	Profile deletion	ProfileID:46	Profile successfully deleted	Profile successfully deleted	Pass
52	user creation	Username : 78	username already exists	username already exists	Pass
78	Profile update	ProfileID: 78, City: "York"	City updated successfully	City updated successfully	Pass
93	Profile creation	Name: "Abcd", Age: 29	Profile added successfully	Profile added successfully	Pass

Integration testing:

Test case ID	Integration point	input	Expected output	Actual output	remarks
ID: 19	Profile Creation - Database	Name: "Abz",	Profile added to the database successfully	Profile added to the database successfully	Data inserted correctly
ID:78	Profile Update- Database	ProfileID: 78, City: "York"	City updated in database successfully	City updated in database successfully	Accurately updates city
ID:46	Profile deletion- Database	ProfileID:46	Error: "Profile not found"	Error: "Profile not found"	Permanently deletes profiles.
NULL	Search Multiple Criteria	Age: 22, City: "London"	Profiles that meet the required criteria are retrieved	Correct profiles retrieved	It correctly retrieves profiles based on multiple criteria
NULL	Search Single Criteria	Gender: “female”	Profiles that meet required criteria are retrieved	Correct profiles retrieved	It correctly retrieves profiles based on criteria

Conclusion:

The program successfully showcased the development of a dating management system that utilises SQL database functionality for easily managing profiles, including creating, reading, updating, and deleting. This Program was built using C#, the system combines object-oriented programming principles with algorithms to ensure functionality, ease of use, and efficiency. The Key features of the system include the ability to create profiles, search for profiles based on various criteria, modify profile details, and delete profiles, all while maintaining consistent and reliable data storage through SQL.

objectives

The primary aim was to develop an easy-to-use program for managing profiles. Users can easily perform CRUD operations, while the SQL database handles data storage. The project focused

on providing accurate data retrieval, efficient search capabilities, and effective error handling for invalid inputs and situations.

Implementation Highlights

1. Data Structures: The project utilised numerous data structures, including Lists for dynamically storing user profiles in memory using SQL and custom objects like PersonProfile to encapsulate attributes related to profiles. These selections promoted modularity, clarity, and adaptability in managing profiles.

2. Algorithms: we used advanced algorithms, such as linear search algorithms, to filter and search profiles according to user-specified criteria, prioritising simplicity and efficiency. Sorting features were integrated using LINQ to enhance data organisation and improve user experience. Additionally, validation logic was included to effectively manage default values, duplicate names, and boundary conditions.

3.Testing and Validation: Rigorous testing was performed to evaluate functional correctness, integration reliability, and performance. This included verifying CRUD operations, boundary testing for edge cases, and manual inspection of the SQL database to ensure data integrity.

Challenges and Solutions

• Dynamic Data Management: One significant challenge was synchronising the profile list with the SQL database. We implemented methods such as LoadProfilesFromDatabase to fetch stored profiles during program initialisation, ensuring data consistency. This approach enabled smooth integration between in-memory data and database storage.

•Validation: Addressing invalid inputs like negative ages or duplicate names necessitated thorough testing and continuous enhancements. We established extensive error-checking systems to guarantee strong validation processes, enhancing system reliability and user experience.

Outcomes

The system provides a versatile and efficient platform for managing user profiles. It effectively addresses practical situations, such as maintaining data across program restarts, offering dynamic filtering options, and providing a comprehensive set of CRUD operations. The integration with SQL enhances scalability and guarantees reliable data management, making the system ideal for small to medium-sized applications.

Reflection and Future Work

This project has successfully achieved its purpose by laying a clear groundwork for a scalable dating management system. Possible improvements may include:

- Implementing a Sorting Function: Adding sophisticated sorting options, such as sorting by number, length of words
- in future projects we will look into exploring alternatives to the Console application for better suitability.

By focusing on these enhancements, the project can transform into a highly secure, scalable, and user-friendly platform capable of meeting complex real-world challenges.