CST2120

Software Engineering Management and Development

Coursework 2 – C# Application Postal Management System

Autumn/Winter term

2024/2025

Date of Submission: 16th April 2025

Team : Yukta Ranjeet Emrith , Rohaj Gokool Oopadhya , Oubaid Maudarbux , Kevan Chinapul , Migisa Kajura

Lab Tutor: Mr. Karel Veerabudren

Table of Content

Introduction

This project presents the development of a Postal Management System using C# Windows Forms and Microsoft SQL Server. The system streamlines postal office operations by enabling the efficient management of clients, postmen, packages, and cash transfers. Each module includes intuitive interfaces for adding, editing, searching, and deleting records. Modular design ensures maintainability and usability (Sommerville, 2016).

To enhance in-memory data handling and performance, custom hash tables are used in modules such as Postman and Package. Hash tables allow for constant-time operations like search, insert, and delete, reducing unnecessary SQL calls and improving responsiveness (Cormen et al., 2009; Goodrich and Tamassia, 2014). The smart search functionality supports autocomplete and case-insensitive partial match filtering, improving record accessibility.

A dynamic file import logic is executed during application startup. The user is prompted to provide a file path or skip import if database records already exist. This ensures that only new OfficerIDs are inserted into the system, maintaining data integrity and usability (Pressman and Maxim, 2020).

Report Layout

This report is structured into the following sections:

- **Design**: Justifies the selection of key data structures such as hash tables for managing data within the postman and package modules. It also presents pseudocode-style algorithmic explanations of core functionalities, including the file import process, hashing logic, and data manipulation workflows.
- **Testing**: Describes the strategy used to validate system functionality and reliability. A test case table is included to summarize validations performed for insertions, updates, deletions, and searches across all modules.
- **Conclusion**: Reflects on the project outcomes, highlighting key achievements, implementation challenges, and system limitations. It also outlines potential improvements for future development iterations.
- **References**: Lists all external resources and academic references cited during the project, both in the code and as a formal Harvard-style bibliography.

System Functionality and User Interface

1. Startup – File Import/Login Portal

Upon launching the application, users are presented with a **Login Portal** that prompts for the path to a .txt file containing postman data (Figure 1).

- **Insert Path**: The user may browse or manually enter the file path. When the "Next" button is clicked, the system:
  - Stores the input path in a global variable (Program.dataFilePath).
  - Validates the file's existence.
  - Loads the content using LoadPostmanData() from Program.cs, which:
    - Connects to the SQL database.
    - Checks for existing OfficerIDs in PostmanTbl.
    - Parses the file line by line and inserts only unique entries, using a hash-based check to avoid duplicates.
  - Upon success, the application redirects to the **Main Menu**.

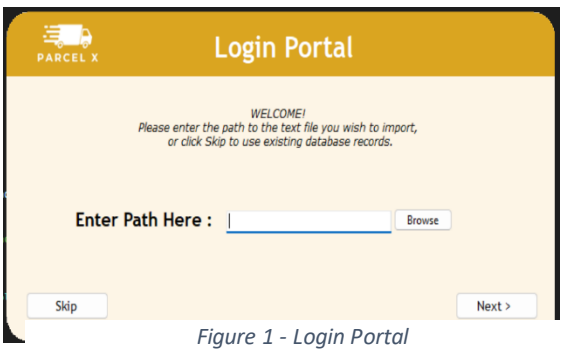

Figure 1 - Login Portal

- **Skip**: If the user does not wish to import data, they may click "Skip". The system then:
  - Executes DatabaseHasPostmanData() to verify if records already exist.
  - If the database contains postman records, the system opens the Main Menu directly.
  - If the database is empty, an error message is shown, requesting the user to provide a valid file path.

2. Main Menu Navigation

After successful login or data validation, the user is redirected to the Main Menu (Figure 2), which offers navigation to four core modules:

- Package Management
- Client Management
- Cash Transfers
- Postman Dashboard

Each section is accessed through visually distinct buttons featuring rounded edges and clear icons for improved accessibility and interface consistency.

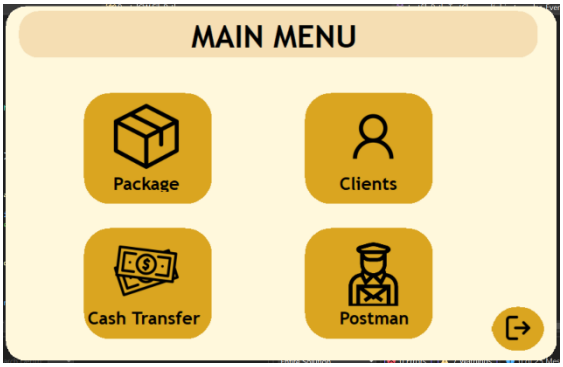

Figure 2 - Main Menu

3. Administrative Modules Overview

The Package, Client, and Cash Transfer modules share a unified structure designed for CRUD operations — allowing users to create, view, update, and delete records through familiar components.

Common Features Across Modules:

- Central DataGridView: Displays all existing records in a tabular format
- Input Sections: Clearly separated fields for form entry (e.g., names, IDs, contact)
- Edit/Save/Delete Buttons: Trigger the corresponding operations with validation
- Dropdowns, calendars, and autocompletion: Improve usability and reduce errors
- Hash Table Integration: Clients, Cash Tranfer and Packages are managed in-memory using custom hash tables to enable fast access and avoid unnecessary database calls

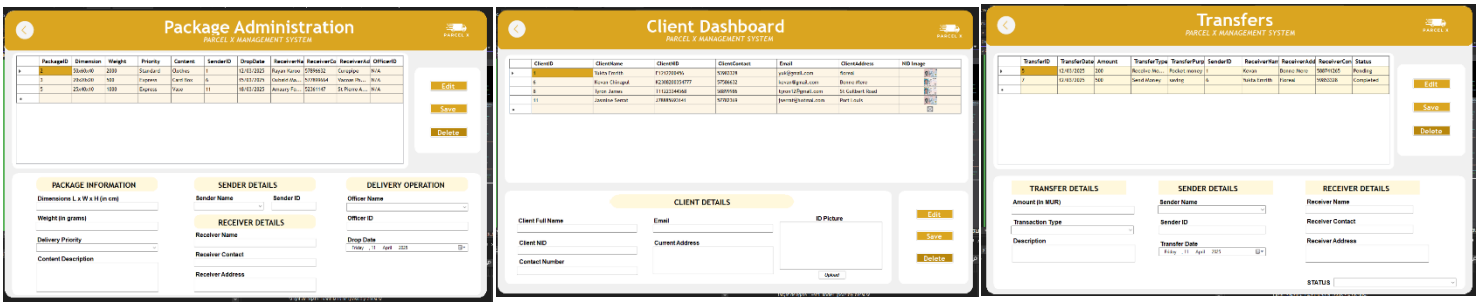

Figure 3 - Package, Client, CashTransfer Dashboard

4. Postman Dashboard

The Postman Dashboard (Figure 4) offers a specialized interface for managing postal officers:

- Users can add new officers, edit existing ones, or delete them
- The interface displays essential fields such as name, address, contact, hire date, and employment type
- Integrated search functionality allows filtering by OfficerID or name, with suggestions based on input

Postman records are first handled using the system's in-memory hash table, and data integrity is enforced through uniqueness checks on OfficerID during file import.

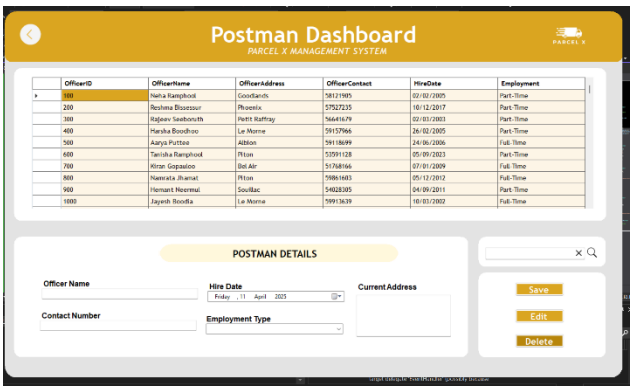

Figure 4 - Postman Dashboard

The Design

Overview

The Postal Management System handles dynamic user data involving postmen, clients, packages, and cash transfers. To reduce database overhead and improve responsiveness during frequent operations like search, edit, and delete, the system uses a **custom hash table (HashTable<T>)** as its in-memory data structure.

Unlike linear lists or trees that offer linear or logarithmic access times, hash tables provide average-case constant time for insertion, deletion, and lookup operations (Cormen et al., 2009; Goodrich and Tamassia, 2014). These performance characteristics make them ideal for a multi-form desktop application requiring real-time interaction.

Justification for Data Structure

A custom hash table (HashTable<T>) was designed and implemented in HashTable.cs to store data temporarily in memory and reduce reliance on SQL queries during UI interactions. It supports:

- **Modular data handling**: Generic design allows reuse for Postman, Client, and Package objects.
- **Fast access**: Provides O(1) average-case time for search, insert, and delete (Sedgewick and Wayne, 2011).
- **Simple collision management**: Uses separate chaining via a list in each bucket for handling hash collisions (McMillan, 2007).
- **Clean separation of UI logic**: Data is only persisted to SQL when explicitly saved, which is ideal for GUI forms (Pressman and Maxim, 2020).

Key Algorithmic Features (with Pseudocode)

❖ Hash Table: Insert

```
Pseudocode: HashTable.Insert(key, value)

CALCULATE index = key MODULO table size

IF bucket at index is EMPTY THEN
    CREATE a new list at index
END IF

FOR each pair in bucket at index DO
    IF key already exists THEN
        UPDATE value
        RETURN
    END IF
END FOR

APPEND (key, value) to bucket at index
```

Time Complexity:

- **Best/Average**: O(1) — key hashes to an empty or single-element bucket.
- **Worst**: O(n) — if many keys hash to the same index (collision-heavy scenario).

Justification: Hash tables perform efficiently under uniform distribution of keys (Cormen et al., 2009). Since our keys (IDs) are numeric and unique, collisions are rare, making O(1) access practical in real-world usage.

❖ Hash Table: Get

```
Pseudocode: HashTable.Get(key)

CALCULATE index = key MODULO table size

IF bucket at index is NOT EMPTY THEN
    FOR each pair in bucket DO
        IF pair.key EQUALS key THEN
            RETURN pair.value
        END IF
    END FOR
END IF

RETURN null
```

Time Complexity:

- **Average**: O(1)
- **Worst**: O(n) — in case of bucket overflow due to poor hash distribution

➢ Used in Clients.cs, Postman.cs, CashTransfer.cs and Package.cs for all fast lookups on ID.

❖ Hash Table: Remove

```
Pseudocode: HashTable.Remove(key)

CALCULATE index = key MODULO table size

IF bucket at index EXISTS THEN
    FOR each pair in bucket DO
        IF pair.key EQUALS key THEN
            REMOVE pair from bucket
            RETURN
        END IF
    END FOR
END IF
```

Time Complexity:

- **Average**: O(1)
- **Worst**: O(n)

Officer File Import Logic

At application startup, the system loads postman data from a .txt file. This functionality ensures that only unique records (based on OfficerID) are inserted into the SQL database — preventing redundancy and preserving data integrity.

This approach is particularly beneficial when postman records are imported from external systems or manually maintained, allowing flexibility in deployments while enforcing strict uniqueness rules.

➢ **Data Structures Involved**

To support this logic, the following in-memory data structures are used:

- **List<int> existingOfficerIDs**: Temporarily stores all OfficerIDs already present in the PostmanTbl database. This list is used to detect and avoid duplicates during the file import.
- **string[] lines**: Holds all lines read from the .txt file, where each line corresponds to a postman record.
- **String.Split()**: Extracts individual officer fields (e.g., ID, name, address, etc.) from each line using a delimiter.

Application Startup Logic

```
Pseudocode: LoadPostmanData()

IF file does not exist THEN
    SHOW error and RETURN

CONNECT to SQL database

IF PostmanTbl has data THEN
    FETCH all OfficerIDs into list

FOR each line in file DO
    SPLIT line into officer fields
    IF OfficerID NOT in list THEN
        INSERT into PostmanTbl
    END IF
END FOR

SHOW confirmation with number of new records added
```

Officer File Loading Logic

```
Pseudocode: Main()

SHOW file path input form

IF user clicks "Insert" THEN
    SET Program.dataFilePath
    CALL LoadPostmanData()
    OPEN Menu form

ELSE IF user clicks "Skip" THEN
    IF database has postmen THEN
        OPEN Menu form
    ELSE
        SHOW error "Database empty"
    END IF

ELSE
    EXIT application
```

Time Complexity:

- File read: O(n)
- Duplicate check with list: O(n)
- Total: $O(n^2)$ worst case for n lines and duplicate checking

  Justification: While $O(n^2)$ may seem inefficient, the operation is only performed once at startup. Given the input file contains ~1000 records, this remains manageable on modern systems. To further optimize, the list can be replaced with a HashSet to reduce lookup time to O(1) (Goodrich and Tamassia, 2014). The current structure was chosen for simplicity and readability, making it easier to maintain and debug.

Search Functionalities (e.g., Postman.cs)

The search functionality allows users to retrieve officer records efficiently using either an exact OfficerID or partial matches on OfficerName.

```
Pseudocode: SearchByIDOrName(input)

INITIALIZE empty result list

IF input is numeric THEN
    GET result using hashTable.Get(input)
ELSE
    FOR each item in hashTable.GetAll() DO
        IF item.Name CONTAINS input THEN
            ADD item to result list
        END IF
    END FOR
END IF

RETURN result list
```

Time Complexity:

The search bar uses the hash table for:

- Direct match: Searches by OfficerID in constant time O(1).
- Partial match: Scans all officer names for text matches (linear time O(n)).

Justification: The system uses a custom in-memory hash table to store all officer records on form load. This hash table offers:

- Fast constant-time lookups for exact OfficerID matches.
- Efficient traversal through values for partial name matches.
- Minimal database queries, boosting UI responsiveness.

The combination of a dictionary-like structure with search logic maximizes both performance and flexibility — ideal for use cases where filtering and speed are critical (Sommerville, 2016).

Testing

Category	Test Case	Test Method	Expected Result	Status
Database Testing	Verify if database (PostmanTbl) has data	ProgramTests.DatabaseHasPostmanData_ShouldReturnTrue	Should return `true` if data exists	✅ Passed
Client Testing	Insert a new client	ClientTests.Client_Insert_ShouldSucceed	Client added successfully	✅ Passed
Client Testing	Search for an existing client	ClientTests.Client_Search_ShouldReturnCorrectClient	Should find and match client name	✅ Passed
Client Testing	Delete an existing client	ClientTests.Client_Delete_ShouldRemoveClientSuccessfully	Client deleted and not found	✅ Passed
Postman Testing	Insert a new postman	PostmanTests.Postman_Insert_ShouldSucceed	Postman added successfully	✅ Passed
Postman Testing	Search for an existing postman	PostmanTests.Postman_Search_ShouldReturnCorrectPostman	Should find and match postman name	✅ Passed
Postman Testing	Delete an existing postman	PostmanTests.Postman_Delete_ShouldRemovePostmanSuccessfully	Postman deleted and not found	✅ Passed
Hash Table Testing	Add a new key-value pair	HashTableTests.HashTable_AddItem_ShouldAddSuccessfully	Key-Value pair added	✅ Passed
Hash Table Testing	Search for an existing key	HashTableTests.HashTable_SearchItem_ShouldReturnCorrectValue	Value retrieved successfully	✅ Passed
Hash Table Testing	Remove an existing key	HashTableTests.HashTable_RemoveItem_ShouldRemoveSuccessfully	Key deleted	✅ Passed
Hash Table Testing	Check if key exists	HashTableTests.HashTable_ContainsKey_ShouldReturnTrue	Should return `true` if key exists	✅ Passed
Menu Testing	Load Menu Form successfully	MenuTests.MenuForm_ShouldLoadSuccessfully	Menu form opens without error	✅ Passed
Menu Testing	Click on Client icon to open Client Form	MenuTests.Menu_ClickClientIcon_ShouldOpenClientForm	Should trigger form load (assumed success)	✅ Passed
Menu Testing	Click on Postman icon to open Postman Form	MenuTests.Menu_ClickPostmanIcon_ShouldOpenPostmanForm	Should trigger form load (assumed success)	✅ Passed
Menu Testing	Click on Logout and handle confirmation box	MenuTests.Menu_LogoutConfirmation_ShouldShowMessageBox	Should show logout confirmation box	✅ Passed

Conclusion

This project successfully delivered a functional and modular Postal Management System using C# Windows Forms and Microsoft SQL Server. The system supports the management of core operations such as postmen, clients, packages, and cash transfers. Key features include the use of a custom hash table for efficient in-memory data handling, dynamic file import logic, and a user-friendly graphical interface with a clear layout and navigation system.

Despite its achievements, the project encountered some limitations. One key challenge was ensuring data consistency between the in-memory hash table and the SQL database. Since data was only written to the database upon clicking Save, there was potential for unsaved changes to be lost if the application closed unexpectedly. Additionally, the system lacked robust input validation and file error handling during the text file import stage, which could cause issues in scenarios involving malformed data.

If the project were to be developed further, improvements would include implementing real-time synchronization between the hash table and database to ensure data is not lost, adding stronger exception handling for file and database operations, and introducing user authentication for security. Future iterations could also improve UI responsiveness and scalability by adopting asynchronous programming patterns and enhancing the structure using MVC or MVVM frameworks for better separation of concerns.

References

- Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2009. *Introduction to Algorithms*. 3rd ed. Cambridge, MA: MIT Press.
- Goodrich, M.T. and Tamassia, R., 2014. *Data Structures and Algorithms in C#*. 2nd ed. Wiley.
- Sedgewick, R. and Wayne, K., 2011. *Algorithms*. 4th ed. Boston: Addison-Wesley.
- McMillan, M.E., 2007. *Data Structures and Algorithms using C#*. Cambridge: Cambridge University Press.
- Pressman, R.S. and Maxim, B.R., 2020. *Software Engineering: A Practitioner's Approach*. 9th ed. New York: McGraw-Hill Education.
- Sommerville, I., 2016. *Software Engineering*. 10th ed. Harlow: Pearson Education.