CST2550

Software Engineering Management and Development Group Project

Car Hire System

2024/25

Date of Submission: Wednesday 16 April 2025.

Student ID Number: M01012616, M01004726, M01012611, M01006629, M01003456

Lab Tutor: Miss Aisha Idoo

Table of Contents

Contents

Table of figures

1. Introduction

This project focused on the design and implementation of a Car Hire System for HorizonDrive LTD using C#.NET and SQL. The core objective was to create a system that supports the management of users (Admins, Customers, Lessors), cars (both company-owned and externally provided), and car bookings. Emphasis was placed on algorithm design and data structure implementation to store and retrieve car and booking information efficiently. This report outlines the data structure design, testing strategy, limitations, and team reflections.

The report is structured into sections covering system design, testing methodology, critical evaluation of outcomes, and recommended future enhancements. Supporting evidence includes pseudocode, time complexity analysis, and test cases.

2. System Overview

2.1 User Roles

1. **Customer:** Access a unified interface. Use the sidebar to browse cars, book a car, manage bookings , update user details, and log out.
2. **Lessor:** Access the same unified interface as customers. Use the sidebar to list cars, view and manage all listed cars, update user details, and log out.
3. **Admin:** Access a separate administrative interface. Features include managing users , managing bookings, handling car maintenance, uploading files, and logging out.

2.2 Features Implemented

- Registration and login (with form validation and role selection)
- Role-specific dashboards with responsive pages
- Car listing and searching options
- Booking with automated receipt generation
- Email notification upon successful booking
- Admin moderation panel for users and bookings
- Password encryption
- Drag-and-drop image upload feature and image slideshow
- Update user profile both through user and admin

4. Design and Architecture

4.1 Justification of Selected Data Structures

1. Hash Table

The hash table is used to store and retrieve key-value pairs efficiently, allowing quick lookups (e.g., by registration number).

- **Implementation:**
  It makes use of **Fibonacci hashing** to distribute hash values more evenly across buckets, which helps avoid clustering (Goodrich and Tamassia, 2011). Common operations like **Insert, Search,** and **Remove** are optimized to run in average **O(1) time**.
- **Justification:**
  This approach provides **rapid data access** and reduces the likelihood of **collisions** (Cormen et al., 2009). It's well-suited for dynamic datasets while maintaining **balanced bucket usage** for consistent performance.

2. Linked List

Linked lists are used within each bucket to handle **collisions,** using a technique called **separate chaining.**

- **Implementation:**
  Each bucket is a LinkedList<KeyValuePair<TKey, TValue>>, which can store multiple key-value pairs when collisions occur (Weiss, 2013).
- **Justification:**
  This prevents **overwriting data** when multiple keys hash to the same bucket. It also allows for **unlimited growth** in a single bucket, making it an effective and flexible collision-handling method(Goodrich and Tamassia, 2011).

3. Array

An array serves as the foundation of the hash table, storing all the buckets.

- **Implementation:**
  A private array named _buckets is used, where each index holds a linked list. This design ensures **constant-time access** to each bucket during operations (Cormen et al., 2009).
- **Justification:**
  It offers the **speed of array indexing** with the **flexibility of linked lists**, resulting in a hybrid solution that supports both **static structure performance** and **dynamic memory allocation** (Weiss, 2013).

4.2 Data Structure Time Complexity Analysis

Structure/Operation	Time Complexity
Hash Table – (Insert, Search, Remove)	**O(1)** (average), **O(n)** (worst case due to collision)
Linked List Traversal	**O(k)** (where k is the length of the linked list in a bucket)
Array Access	**O(1)** (constant time for indexed access)

4.3 Justification of Selected Algorithms

1. Type Chaining

It incorporates generic types (TKey and TValue) for diverse key-value combinations and uses LinkedList<KeyValuePair<TKey, TValue>> for collision resolution.

• **Justification**:

Type chaining enhances the adaptability of the hash table across different applications while maintaining robust collision-handling techniques (Weiss, 2013).

2. Resizing Mechanism

It dynamically expands the table's capacity when the load factor exceeds 0.75. The new capacity is set to the next prime number, and all key-value pairs are redistributed to the updated buckets.

• **Justification**:

The resizing mechanism minimizes collisions and ensures consistent performance as the dataset grows (Cormen et al., 2009).

4.4 Key Algorithms – Pseudocode Representation

1. Hashing Function

This method calculates and returns the hash index for a given key using Fibonacci hashing for even distribution.

Pseudocode:

```
METHOD GetHash(key)
  // Step 1: Generate a numerical representation for the key.
  hash = key.GetHashCode()

  // Step 2: Apply Fibonacci scaling to spread values evenly across buckets.
  product = hash * 0.618033988749895

  // Step 3: Extract the fractional portion of the scaled hash.
  fractionalPart = product - FLOOR(product)

  // Step 4: Map the fractional value to the bucket range.
  RETURN FLOOR(_capacity * fractionalPart)
END METHOD
```

Figure 1: Hashing Function Pseudocode

2. Insertion into HashTable

This method calculates the hash index for a given key. It ensures the key is mapped to a valid bucket index by using Fibonacci hashing for even distribution.

Pseudocode:

```
METHOD GetHash(key)
  // Step 1: Generate a numerical representation for the key.
  hash = key.GetHashCode()

  // Step 2: Apply Fibonacci scaling to spread values evenly across buckets.
  product = hash * 0.618033988749895

  // Step 3: Extract the fractional portion of the scaled hash.
  fractionalPart = product - FLOOR(product)

  // Step 4: Map the fractional value to the bucket range.
  RETURN FLOOR(_capacity * fractionalPart)
END METHOD
```

Figure 2: Insertion in HashTable Pseudocode

3. Search into HashTable

This function searches for a key in the hash table and retrieves and returns its associated value if found.

Pseudocode:

```
METHOD Search(key): value
  // Step 1: Calculate the bucket index for the key using the hashing function.
  index = GetHash(key)

  // Step 2: Check if the bucket exists at the calculated index.
  IF _buckets[index] IS NOT NULL THEN
    // Step 3: Iterate through the bucket to locate the key.
    FOR EACH pair IN _buckets[index] DO
      IF pair.Key == key THEN
        RETURN pair.Value     // Return the value if the key matches.
      END IF
    END FOR
  END IF

  // Step 4: Return the default value if the key is not found in the hash table.
  RETURN DEFAULT VALUE
END METHOD
```

Figure 3: Search in HashTable Pseudocode

4. Deletion from HashTable

This function removes a key-value pair from the hash table and returns a boolean indicating success.

Pseudocode:

```
METHOD Remove(key): BOOLEAN
  // Step 1: Calculate the bucket index for the given key.
  index = GetHash(key)

  // Step 2: Check if a bucket exists at the calculated index.
  IF _buckets[index] IS NOT NULL THEN
    // Step 3: Search for the key in the bucket.
    node = FIND node IN _buckets[index] WHERE node.Key == key
    IF node EXISTS THEN
      // Step 4: Remove the key-value pair from the bucket.
      _buckets[index].Remove(node)

      // Step 5: Decrement the hash table's size counter.
      _size--

      // Step 6: Return true to indicate successful removal.
      RETURN TRUE
    END IF
  END IF

  // Step 7: Return false if the key is not found in the table.
  RETURN FALSE
END METHOD
```

Figure 4: Deletion from HashTable

5. Resize of HashTable

This procedure expands the hash table's capacity and redistributes key-value pairs to maintain efficiency and reduce collisions.

Pseudocode:

```
METHOD Resize()
  // Step 1: Calculate the new capacity by doubling the current capacity
  //       and finding the next prime number.
  newCapacity = GetNextPrime(_capacity * 2)

  // Step 2: Create a new bucket array with the updated capacity.
  newBuckets = ARRAY of LinkedList[newCapacity]

  // Step 3: Iterate through all existing buckets to rehash key-value pairs.
  FOR EACH bucket IN _buckets DO
    IF bucket IS NOT NULL THEN
      // Step 4: Rehash each key-value pair and move it to the new bucket.
      FOR EACH pair IN bucket DO
        newIndex = GetHash(pair.Key)        // Get new index using hash function
        IF newBuckets[newIndex] IS NULL THEN
          newBuckets[newIndex] = NEW LinkedList() // Initialize bucket if needed
        END IF
        newBuckets[newIndex].AddLast(pair)  // Add pair to the new bucket
      END FOR
    END IF
  END FOR

  // Step 5: Replace the old bucket array with the new bucket array.
  _buckets = newBuckets

  // Step 6: Update the hash table's capacity to the new capacity.
  _capacity = newCapacity
END METHOD
```

Figure 5: Resizing of the HashTable Pseudocode

4.4 System Architecture Overview

- **Frontend**: WinForms Desktop Application (Wireframes are in the appendix).
- **Backend**: C#.NET for business logic and operations.
- **Database**: SQL Server (Database structure created in Lucidchart and included in the appendix).
- **Architecture Pattern**: SOLID principles for modularity and maintainability.
- **Methodology**: Agile methodology was followed to ensure iterative development and flexibility.

5. Testing and Evaluation

5.1 Statement of Testing Approach

The testing approach employs diverse techniques to ensure software robustness and validate functionalities. Below are the strategies implemented:

1. **Black-box Testing**:
   - Evaluated the external behavior of the application by simulating user interactions in the **WinForms UI**.
   - Ensured the system behaved as expected without analyzing its internal workings.
2. **Unit Testing**:
   - Focused on validating individual methods and components using **MSTest** as the primary framework.
   - Examples include hash table functionalities (SearchTests.cs, RemoveTests.cs) and validations like license file uploads (OptionsPreferencesTests.cs).
3. **Edge Case Testing**:
   - Covered scenarios like empty field submissions (SignupTests.cs), handling non-existent keys in operations (SearchTests.cs), and input validation to prevent issues like SQL injection.
   - Ensured the application gracefully handled unexpected inputs or edge cases.
4. **Integration Testing**:
   - Validated the interaction between modules such as database operations and form submission workflows.
   - Tests such as HashTableResizeTests.cs ensured components worked cohesively under realistic load conditions.
5. **Debugging Techniques**:

   Leveraged advanced debugging tools in Visual Studio for runtime optimization:

   - **Breakpoints** for step-by-step code execution.
   - **Watch Windows** for runtime variable inspection.
   - **Call Stack** for tracking execution flow.

Memory debugging and performance profiling further optimized efficiency during testing.

6. **Validation of Use Cases** (replacing Test Data and Coverage):
    - Ensured all key use cases, such as file uploads (OptionsPreferencesTests.cs) and car management workflows (RemoveTests.cs, HashTableResizeTests.cs), were fully tested.
    - Focused on scenarios encountered in real-world workflows to provide comprehensive validation.

5.2 Table of Test Cases

.cs File Name	Feature	Input	Expected Output	Actual Outcome	Pass/Fail
InsertTests.cs	Insert Key-Value Pair	Key: 1, Value: "Car"	Key inserted successfully	As expected	Pass
SearchTests.cs	Search Existing Key	Key: 99, Value: "Mauritius"	"Mauritius"	As expected	Pass
SearchTests.cs	Search Non-existent Key	Key: 123	Returns null	As expected	Pass
RemoveTests.cs	Remove Key-Value Pair	Key: "car", Value: "Toyota"	Key removed successfully	As expected	Pass
HashTableResizeTests.cs	Hash Table Resizing	Keys: {1, 2, 3}	Items preserved post-resizing	As expected	Pass
LinkedListIterationTests.cs	Linked List Iteration	Items: {Rumaisa, Dhruv, Aayush}	Items retrieved in order	As expected	Pass
OptionsPreferencesTests.cs	Upload License	File: "C:\\DriverLicense.jpg"	File uploaded successfully	As expected	Pass
SignupTests.cs	Validate Empty Fields	Fields: "", "", ""	Validation error displayed	As expected	Pass

6. Conclusion

6.1 Summary of Work Completed

The Car Hire System project adhered to a structured software development life cycle, starting with detailed planning and design, including wireframes and an ERD to model database relationships. Developed with C# WinForms, the system prioritized responsiveness, featuring drag-and-drop image uploads, automatic receipt generation, email notifications upon successful bookings, and robust validation protocols for user interaction. The backend ensured smooth operations for bookings, authentication, and role-based access control for Admins, Customers, and Lessors. Agile methodology enabled iterative progress, regular feedback, and adaptability, while Microsoft unit testing validated core components to ensure reliability and correctness.

6.2 Limitations and Critical Reflection

The project successfully delivered core functionalities but faced limitations arising from technical, time-related, and methodological constraints. Technically, the use of WinForms limited scalability, and the omission of CAPTCHA and API integrations for car image descriptions was caused by financial restrictions and limited experience with external APIs. Time constraints, worsened by a preponed deadline, forced rushed development phases, resulting in reduced feature depth and incomplete advanced testing. Methodologically, Agile's iterative processes, while effective, required frequent revisions and meetings that seemed time-consuming, adding pressure within the tight timeline. These challenges emphasize the importance of careful planning, better resource allocation, and optimized time management for future projects.

6.3 Recommendations for Future Projects

To address the limitations highlighted, future projects should prioritize the adoption of modern frameworks to enhance scalability and usability. Testing should be integrated throughout the development lifecycle to detect and resolve issues efficiently. Improved collaboration through standardized coding practices and effective version control can minimize conflicts during development. Additionally, granular Agile planning and balanced sprint prioritization can ensure adequate time allocation for both core and advanced features, enabling smoother and more efficient development processes.

Reference List:

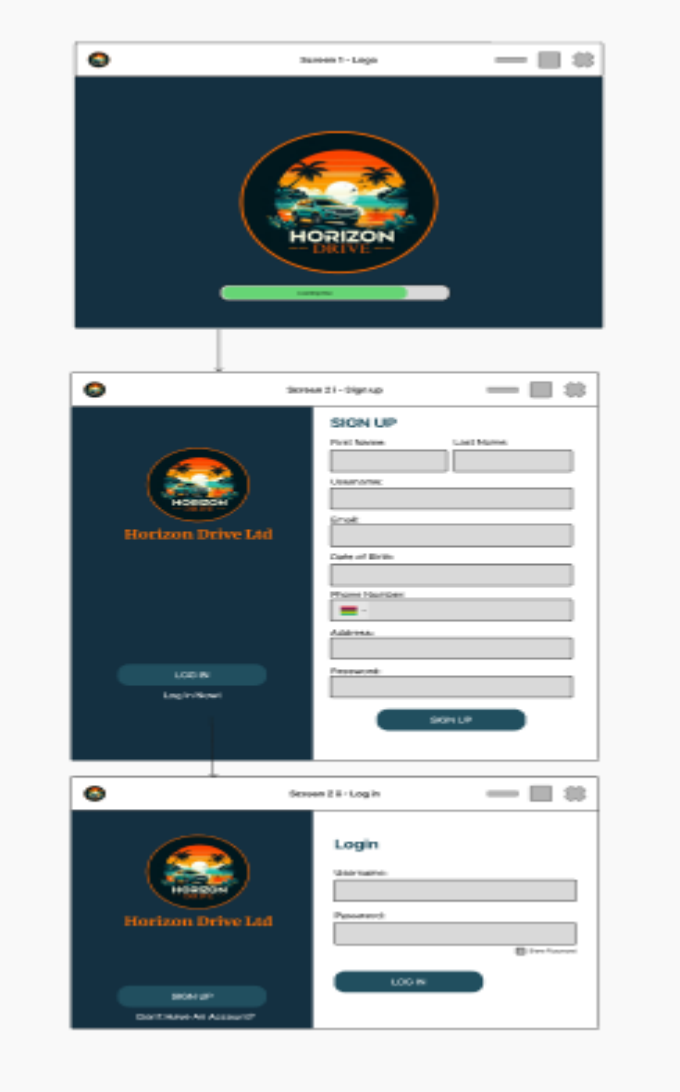
- Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. (2009) *Introduction to Algorithms*. 3rd edn. MIT Press. Available at: https://mitpress.mit.edu/ (Accessed: 14 April 2025).
- Goodrich, M.T. and Tamassia, R. (2011) *Data Structures and Algorithms in C#*. Wiley. Available at: https://www.wiley.com/ (Accessed: 14 April 2025).
- Weiss, M.A. (2013) *Data Structures and Algorithm Analysis in C#*. Pearson. Available at: https://www.pearson.com/ (Accessed: 14 April 2025).

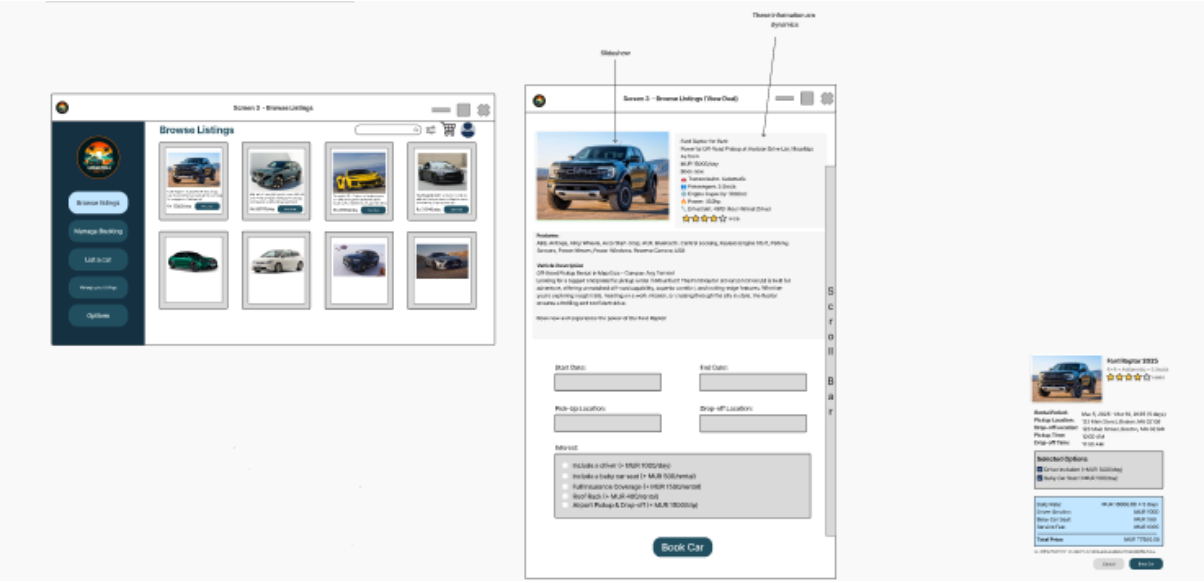
Appendices:

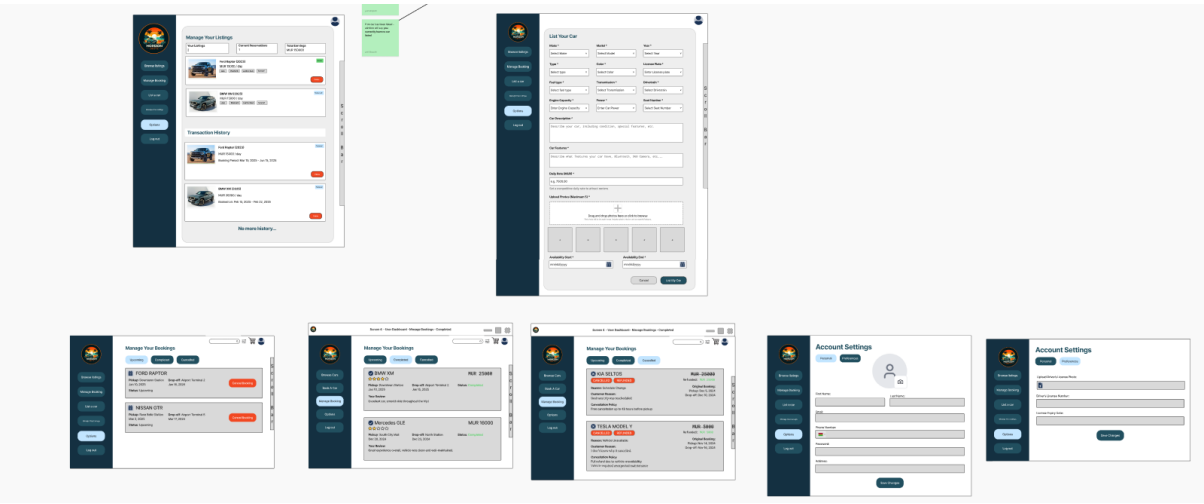
Appendix A: GitHub Repository Link- GitHub - Dhruv-2204/Horizon_Drive_LTD: CST2550 - Software Engineering CW

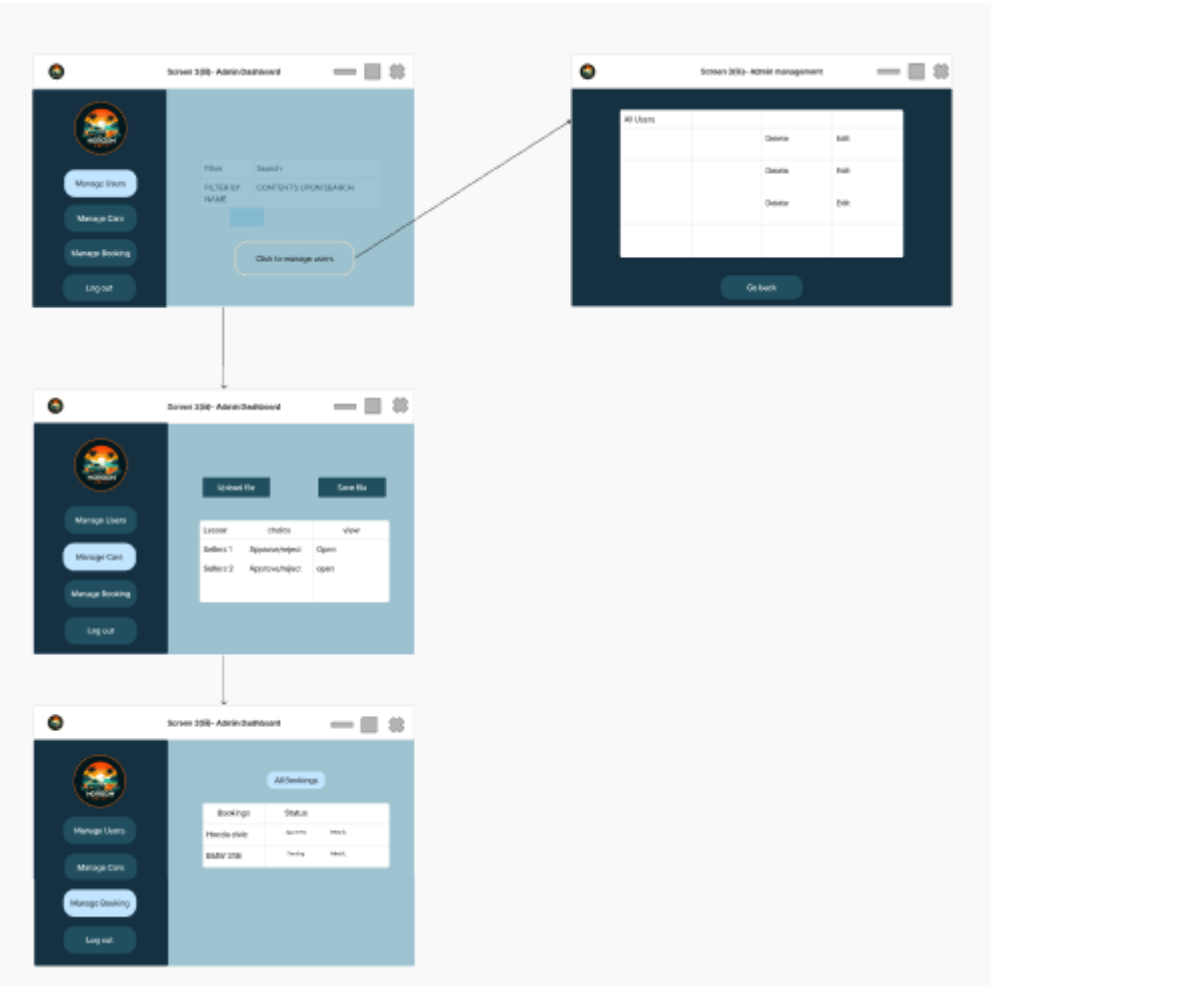
Appendix B: Figma UI Design Screenshots- Horizon Drive Ltd – FigJam

Figure 6: Figma wireframe screenshots






Manage Your Listings
Transaction History
List Your Car
Manage Your Bookings
Account Settings


Manage Users
Manage Cars
Manage Booking
Log out
Click to manage users.
All Users
Go back
Upload file
Save file
Lessor
chains
view
All bookings
Booking
Status

Appendix C: Lucid ERD Diagram Screenshots- [ERD FOR HORIZON DRIVE: Lucidchart]

Figure 7: ERD Diagram from Lucid Screenshot