Report: Car Sharing Management System

Developed by:

- **Vesisss** (Giuseppe Santaniello)
- **Erapty** (Marco Tucci)
- **rob426** (Roberto Pucci)

---

1. Motivation Behind the Choice of Abstract Data Types (ADTs)

While designing the Car Sharing Management System, the choice of Abstract Data Types (ADTs) played a pivotal role in ensuring both the maintainability and performance of the application. The system is aimed at managing vehicle rentals, user registrations, and historical bookings in a scalable and organized manner.

To achieve this, we decided to adopt the following ADTs:

- **Hash Table (HashTable):** Used for storing and retrieving user and vehicle records efficiently. By using unique keys (such as username or license plate), we ensured that lookups, insertions, and deletions could be performed in average constant time. This is crucial given the potentially large number of users and vehicles in a real-world scenario.

- **Linked List (List):** Employed to manage bookings. We opted for a simple, dynamically allocated singly linked list which allows flexible insertion of new bookings and maintains a sequential ordering (chronologically, if desired). It also supports traversal operations for filtering bookings (e.g., upcoming, ongoing, past).

These choices support:

- Fast access and search capabilities via keys
- Dynamic memory allocation with no predefined size limits
- Modular design and separation of concerns
- Clear encapsulation and adherence to information hiding principles

By building these ADTs internally, we retained full control over memory and behavior, aligning the system closely with the specific requirements of the domain.

---

2. How to Use and Interact with the System

After saving all the project files to a local folder on your computer, follow these steps to compile and execute the system:

1. **Open a terminal** and navigate to the folder containing all source files and the Makefile.

2. Use the following commands:

- **`make run`**
  Compiles and launches the main application, allowing interaction with the car sharing system via a terminal-based menu (for both clients and administrators).

- **`make test`**
  Compiles and executes the automated test suite. This will run predefined test cases to verify booking creation, cost calculations, and booking history accuracy. Results will be shown in the terminal and saved to a `result.txt` file.

- **`make clean`**
  Removes all temporary or generated files such as `.o` object files, `.exe` executables, and `.txt` output files created during the build or test process. Use this command to reset the working directory.

The Car Sharing Management System offers a console-based interface divided into two main user roles: **Administrator** and **Client**.

- **Administrator functionalities include:**
    - Adding or removing users and vehicles
    - Viewing all users,vehicles and bookings in the system
- **Client functionalities include:**
    - Registering or logging in to the system
    - Viewing available vehicles filtered by location
    - Creating a new booking specifying vehicle, start time, and duration
    - Viewing personal booking history (including cost and status)

Interaction workflow:

1. Upon startup, the system loads users, vehicles, and bookings from respective text files.
2. The user selects a role and proceeds through a menu of numbered options.
3. Inputs are collected via standard input; outputs and confirmations are printed on screen.
4. Bookings are created after verifying vehicle availability and cost is calculated based on the selected duration and hourly rate.
5. Upon exit or specific triggers, data is saved back to persistent storage (`users.txt`, `vehicles.txt`, `bookings.txt`).

This structure ensures both data persistence and user session continuity across multiple runs.

---

3. System Design Overview

The application architecture is modular, facilitating both maintainability and collaborative development. Each module is responsible for a specific functionality and communicates via well-defined interfaces.

Modular components:

- `main.c`: Entry point and application control flow
- `admin.c` / `client.c`: Interfaces for administrator and client functionalities
- `user.c`: User registration, lookup, and authentication
- `vehicle.c`: Vehicle registration, listing, and filtering
- `booking.c`: Creation, status evaluation, and formatting of booking records
- `storage.c`: Load and save routines from and to text files
- `hash.c` / `list.c`: Internal ADT implementations for hash tables and linked lists
- `timeutils.c`: Utility functions for date/time conversions (e.g., to/from UNIX timestamps)

Each component includes an associated header file to expose necessary functionality without leaking internal representations, preserving encapsulation.

---

4. Syntactic and Semantic Specification of the ADTs

This section presents the syntactic interface and semantic behavior of the core Abstract Data Types and modules used in the project. All ADTs follow strict encapsulation rules, using opaque pointers and accessor functions to avoid exposing internal structures.

4.1 Hash Table ADT (`hash.h`)

Purpose: Efficient storage and retrieval of key-value pairs, used for `User` and `Vehicle` mappings.

- **`insertHash(HashTable* ht, const char* key, void* value)`**
  Pre: `ht` is a valid pointer; `key` is non-NULL and unique in context
  Post: Associates `value` with `key`, replacing any previous value
  Effect: Insertion or update of an entry in the hash table

- **`findHash(HashTable* ht, const char* key)`**
  Returns: Pointer to value associated with `key`, or NULL if not found
  Effect: No modification to the table

- **`removeHash(HashTable* ht, const char* key)`**
  Pre: `ht` and `key` must be valid
  Post: Removes key and associated value from table (if exists)
  Effect: Memory ownership remains with caller

- **`forEachHash(HashTable* ht, void (*callback)(const char*, void*, void*), void* userData)`**
  Effect: Applies `callback` to every key-value pair; `userData` is passed through

4.2 List ADT (`list.h`)

Purpose: Ordered container for storing `Booking` entries sequentially.

- **`createList()`**
  Returns: New empty list
  Post: List has zero elements

- **`insertList(List list, void* data)`**
  Effect: Adds `data` to the end of the list

Post: `getSize(list)` is incremented by 1

- **`getSize(List list)`**
  Returns: Integer number of elements currently in list
  Effect: No change to list state

- **`getItem(List list, int index)`**
  Pre: `0 <= index < getSize(list)`
  Returns: Pointer to the element at position `index`
  Effect: No modification

4.3 Vehicle ADT (`vehicle.h`)

Purpose: Models a rentable vehicle with attributes such as plate, type, cost, and location.

- **`createVehicle(...)`**
  Returns: New dynamically allocated vehicle
  Post: All fields are initialized and accessible via getters

- **`insertVehicle(HashTable* table, Vehicle* v)`**
  Effect: Registers vehicle in the vehicle hash table using plate as key

- **Accessors:**
  All vehicle properties are accessed via:
  - `getVehiclePlate(v)`
  - `getVehicleCost(v)`
  - `getVehicleSeats(v)`, etc.
    Effect: All return `const` or `int` values without modifying state

- **`printAvailableVehiclesAt(...)`**
  Effect: Filters and prints vehicles available in a given time window using booking data

4.4 User ADT (`user.h`)

Purpose: Manages user identity, login credentials, and types (client/admin).

- **`createUser(const char* username, const char* password, int isAdmin)`**
  Returns: New user instance

Post: Stores login data and user role

- **`checkPassword(User* u, const char* password)`**
  Returns: `true` if passwords match
  Effect: Stateless comparison

- **`isUserAdmin(User* u)`**
  Returns: Boolean flag if user has admin privileges

- **Getters:**

  - `getUsername(User* u)`
    Effect: No modification

4.5 Client Module (`client.h`)

Purpose: Exposes interaction logic for authenticated client users

- **`clientMenu(...)`**
  Pre: Valid login session and data loaded
  Effect: Allows clients to view, select, and book vehicles through menu interface
  Post: May trigger new bookings and file updates

4.6 Booking ADT (`booking.h`)

Purpose: Represents a reservation by a user for a vehicle at a specific time.

- **`createBooking(username, vehicle, start, duration)`**
  Returns: New `Booking` instance
  Post: Fields `start`, `end`, `vehicle`, `user` are set
  Side Effect: `totalCost` is computed based on vehicle rate

- **`getBookingStatus(Booking* b, long now)`**
  Returns: Enum value: `PAST`, `ONGOING`, or `UPCOMING`
  Effect: Stateless evaluation of booking time window

- **Encapsulation enforced:** All fields (e.g., `totalCost`, `discounted`) are managed through accessors only. Direct access is disallowed outside `booking.c`.

---

5. Test Case Rationale

To validate system correctness and behavior across core use cases, we designed and executed the following automated test scenarios:

- **TestCase1 - Booking Creation:**
    - Objective: verify that the system correctly registers a new booking if the selected vehicle is available.
    - Inputs: predefined users and vehicles, booking requests
    - Validation: output file matches the expected `_oracle.txt`
- **TestCase2 - Rental Cost Calculation:**
    - Objective: confirm that booking cost is correctly computed based on vehicle hourly rate and duration
    - Validation: verify cost output against known values
- **TestCase3 - Booking History Accuracy:**
    - Objective: ensure that the system correctly lists all bookings for a given user, including correct status and timing
    - Additional check: verify output order matches expected format

Each test is automated via `run_test_suite.c`, which reads test definitions from `test_suite.txt`, executes them, and reports results.
