Car Rental Simulator
A Memory-Hierarchy Management Case

Jennifer Vicentes
Department of Computer Science, Texas Tech University

April 26, 2025

Abstract

This report presents the design, implementation, and evaluation of a Java-based car rental simulator that models a two-level memory hierarchy: *lots* as secondary storage and *shops* as main memory. The first component, `LotManager`, maintains persistent text files for each lot, while the second, `RentalShop`, provides an interactive command-line (and optional GUI) shop interface. I detail file formats, concurrency control via file locking, state persistence (binary snapshots and human-readable dumps), command semantics, class design, testing scenarios, and extra credit UI integration. All design decisions, including dual-format serialization and global rental registries, are thoroughly justified. Finally, I disclose AI-assisted code generation and modifications in depth.

1 Introduction

Modern operating systems rely on layered memory hierarchies to balance speed, capacity, and persistence. This project uses an automotive rental analogy to illustrate how entities (vehicles) move between:

- **Secondary storage**: represented by *lots*, each persisted as a text file on disk.
- **Main memory**: represented by active *rental shops* maintaining in-memory inventories and processing customer operations.

By implementing two cooperating Java applications—`LotManager` and `RentalShop`—students gain hands-on experience with file-based storage, concurrency control, object serialization, and basic GUI design, achieving the following learning outcomes:

1. Differentiate memory management and data storage techniques.
2. Implement system-level file I/O with concurrency safeguards.
3. Design and document modular software components.
4. Develop a simple but complete desktop UI.

2 Project Overview

The project comprises two programs:

`LotManager` A command-line utility (and Swing-based GUI) to create and maintain `<lotName>.txt` files, each listing vehicles (license plate, type, odometer). It supports adding sedans, SUVs, vans, and removing vehicles by plate, ensuring unique Costa Rican plates via a global registry file.

`RentalShop` An interactive CLI (and Swing-based GUI) that simulates a customer-facing rental shop. It initializes with a location, available parking spaces, and a list of lot names. Customers issue `RENT`, `RETURN`, `LIST`, and `TRANSACTIONS` commands. The program fetches vehicles from lots or inventory, applies discounts, updates odometers, logs transactions, and persists state in both binary and human-readable formats.

Multiple instances of `RentalShop` may run concurrently, coordinating through file locking and a shared `rented_registry.txt`, allowing any shop to return any vehicle.

3 Detailed Requirements and Simulation Mechanics

3.1 Lot Manager

- **Flags:** `-lot-name=<name>`, `-add-sedan=<n>`, `-add-suv=<n>`, `-add-van=<n>`, `-remove-vehicle=<plate>`.
- **Storage:** `<name>.txt` holds lines `PLATE,TYPE,KILOMETERS`.
- **Plate Generation:** Random CR format (AAA-999), checked against `plates_registry.txt` to ensure global uniqueness.
- **Behavior:** On each invocation, read existing file (or create it), apply additions/removals, then write back updated inventory.

3.2 Rental Shop

- **Flags:** `-location=<city>`, `-spaces-available=<n>`, `-lots=<lot1,lot2,...>`.
- **Initialization:** If `city.txt` exists, load prior state (binary). Otherwise, take flags to set up `spacesAvailable` and sources.
- **Commands:**

  `RENT <TYPE>` Allocate from shop inventory if available, else fetch from any lot (10% discount).

  `RETURN <PLATE> <KM>` Update odometer, compute charge ($1/km minus discount), log transaction, ensure $>=$ 2 empty slots (may push highest-km vehicle back to lot).

  `LIST` Print current inventory, rented out, empty slots, earnings.

`TRANSACTIONS` Display all return transactions, total earnings, and total lost due to discounts.

- **Concurrency:** Lock each lot file during read/write to prevent inter-shop conflicts. Use a shared `rented_registry.txt` to track vehicles in transit across shops.
- **Persistence:**
  - `.ser`: Java serialization of entire shop state for fast reload.
  - `.txt`: Human-readable dump (inventory, rented out, transactions) for instructor inspection.

4 Folders Structure

src/main/java/carrental/ `LotManager.java`, `RentalShop.java`, `CarRentalUI.java`, `Vehicle.java`, `Transaction.java`.

tests/lot/ Shell scripts and input files for lot scenarios.

tests/shop/ Shell scripts (`rental-scenarios.sh`), command lists (`shop1-commands.txt`, `shop2-commands.txt`).

target/classes/ Compiled '.class' files for execution.

5 Compilation and Execution Instructions

1. **Compile all Java sources:**

   ```
   cd car-rental-simulator
   mvn compile
   ```

   Then, in order to run the LotManager, RentalShop and CarRentalUI also go to:

   ```
   cd target
   cd classes
   ```

   All the `*.txt` and `*.ser` are going to appear under the `classes/` folder.

2. **Run LotManager example:**

   ```
   java carrental.LotManager \
     --lot-name=Central --add-sedan=2 --add-suv=1
   ```

3. **Run RentalShop CLI:**

   ```
   java carrental.RentalShop \
     --location=SanJose --spaces-available=5 --lots=Central,North
   ```

4. **Run Swing GUI (extra credit):**

```
java carrental.CarRentalUI
```

5. **Execute automated tests:** For this you have to do it under the root-folder: cd car-rental-simulator

   ```
   bash run.sh
   ```

   Test outputs and generated `*.txt` files will appear under `tests/`.

6 Class Diagram Design and Explanation

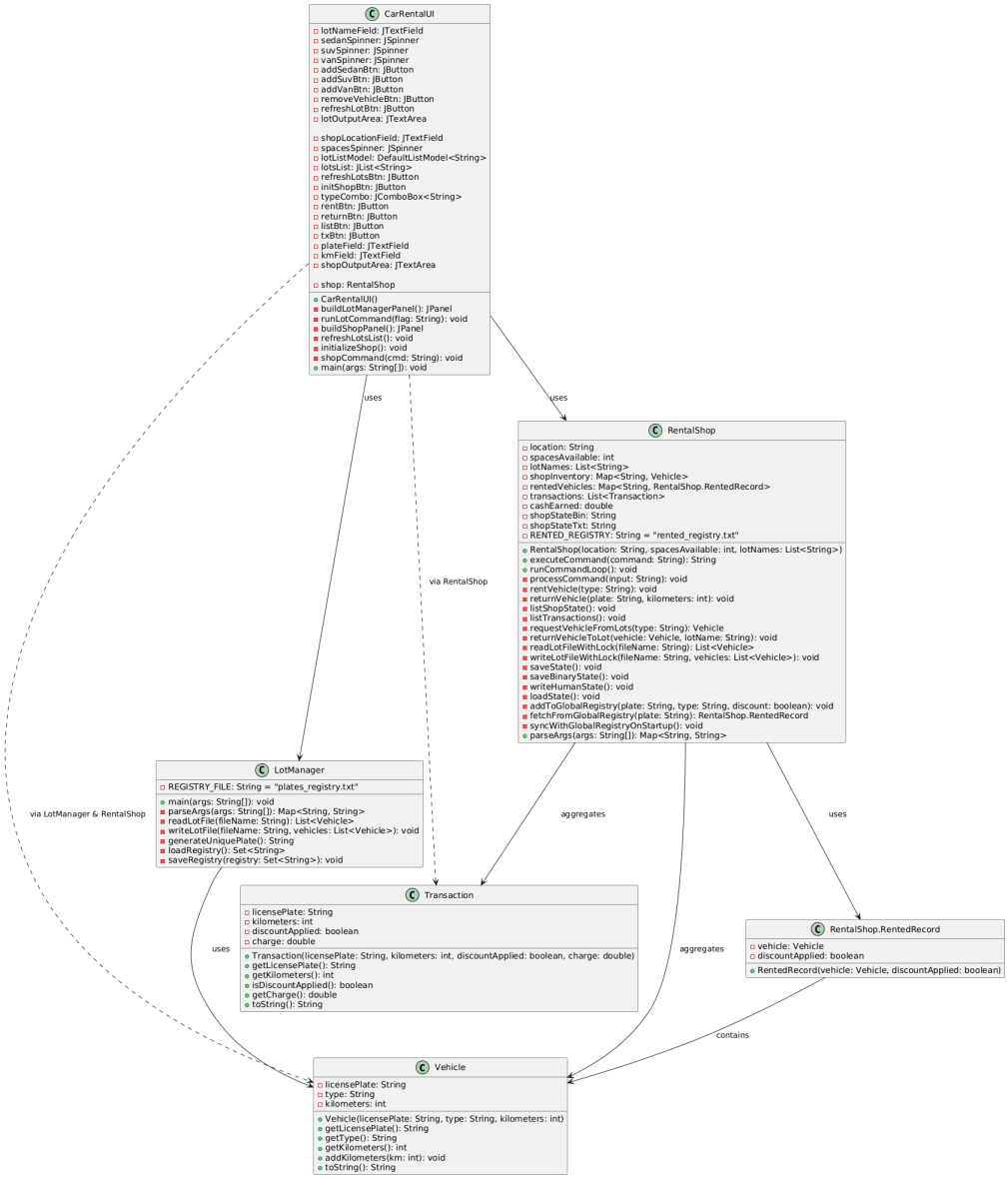

Figure 1: UML Class Diagram for Car Rental Simulator

Overview of responsibilities:

- `LotManager`: CLI argument parsing, lot file maintenance, plate registry management.
- `RentalShop`: Core business logic, state management, concurrency control, command processing, persistence.
- `Vehicle`: Encapsulates plate, type, odometer; serializable for persistence.
- `Transaction`: Records return details; serializable for persistence.
- `RentedRecord` (inner class): Associates a vehicle with discount flag while rented out.
- `CarRentalUI`: Swing GUI wrapper invoking `RentalShop` and `LotManager` for each action.

7 Class and Method Descriptions

7.1 LotManager.java

The `LotManager` class is responsible for maintaining the persistent storage of vehicle inventories in disk-backed text files ("lots"). It follows a typical command-line utility pattern:

- **Main Method:** Parses command-line flags, validates the required `-lot-name`, then delegates to methods that read the lot file, process additions and removals, and write the updated inventory back to disk.
- `parseArgs(...)`: Converts an array of `String` arguments into a `Map<String,String>` of flag names to values, allowing flags with and without explicit values.
- `readLotFile(...)`: Opens (or creates) the specified `.txt` file, reads each line as a comma-separated record (plate, type, kilometers), and instantiates corresponding `Vehicle` objects.
- `writeLotFile(...)`: Serializes the current `List` back to the lot file, overwriting previous contents in a human-readable CSV format.
- `generateUniquePlate()`: Produces a new Costa Rican license plate of the form AAA-3DigNum, ensuring uniqueness by consulting a global registry file protected by `loadRegistry()` and `saveRegistry()`. A `Random` generator selects letters and digits until an unused plate is found.
- `loadRegistry()/saveRegistry()`: Manage the `plates_registry.txt`, reading existing plates into a `Set` on startup and persisting new entries after generation. These methods handle `IOExceptions` gracefully, printing errors to `System.err`.

This design encapsulates all lot-level operations in a single class, emphasizing modularity and reuse of file I/O logic.

7.2 RentalShop.java

The `RentalShop` class models the main-memory cache layer, providing an interactive interface for customers to rent and return vehicles. Key responsibilities include state management, concurrency control, and command processing.

Core Fields

- `location, spacesAvailable, lotNames`: Define the shop's identity, capacity, and source lots.
- `shopInventory, rentedVehicles, transactions`: In-memory data structures storing available vehicles, active rentals (with discount flags), and completed return transactions.
- `cashEarned`: Accumulates revenue across returns.
- `shopStateBin/shopStateTxt`: Filenames for binary snapshots (Java serialization) and human-readable state dumps.

State Initialization and Synchronization

- `RentalShop(...)` Constructor: On startup, checks for an existing `.txt`. If found, loads prior state via `loadState()` and synchronizes rentals against the global `rented_registry.txt` using `syncWithGlobalRegistryOnStartup()`, which acquires a shared file lock to prevent concurrent writes while reading.
- `initializeInventory()`: If starting fresh, fetches one vehicle of each type from the lots using `requestVehicleFromLots()`, demonstrating lazy loading of secondary storage into main memory.

Command Loop and Output Capture

- `runCommandLoop()`: Reads console input and routes commands to `processCommand(...)`, persisting state after each invocation.
- `executeCommand(...)`: Wraps `processCommand` by redirecting `System.out` to a `ByteArrayOutputStream`, enabling programmatic capture of output for testing or GUI display.

Rental and Return Logic

- `rentVehicle(type)`: Attempts to find a vehicle in `shopInventory` first; if none, invokes `requestVehicleFromLots(type)` to pull from a lot with a 10
- `returnVehicle(plate, km)`: Looks up the `RentedRecord` locally or fetches via `fetchFromGlobalRegistry(...)`, updates odometer, computes charge ($1/km minus 10% if discounted), logs a `Transaction`, and ensures parking capacity by returning the highest-km vehicle back to a lot when empty slots fall below a threshold.

File Locking for Lot Coordination

- `readLotFileWithLock(...)`/`writeLotFileWithLock(...)`: Use `RandomAccessFile`, `FileChannel`, and `FileLock` to safely read or overwrite lot files in concurrent scenarios.

Persistence of Shop State

- `saveState()`: Delegates to `saveBinaryState()` (fast reload via Java `ObjectOutputStream`) and `writeHumanState()` (plain-text dump for instructor review).
- `loadState()`: Ingests the binary snapshot back into in-memory fields, using `ObjectInputStream`. Fields are reconstituted in the same order they were serialized.

7.3 CarRentalUI.java

The `CarRentalUI` class unifies both the lot management and rental shop interfaces into a single Swing application, improving usability and consolidating functionality. It uses a `JTabbedPane` with two tabs—"Lot Manager" and "Rental Shop"—to keep concerns clearly separated while sharing event-driven logic.

In the **Lot Manager** tab, text fields and spinners allow the user to specify a lot name and the number of sedans, SUVs, or vans to add at once. Buttons trigger operations that invoke `LotManager.main(...)` with appropriate flags, capture the console output, and then display both the log and the updated contents of the corresponding `.txt` file. This design leverages the existing command-line logic without duplicating file I/O code in the GUI, while automatic list refreshing ensures the rental shop always sees up-to-date lot names.

In the **Rental Shop** tab, the user configures shop location, available parking spaces, and selects one or more lots from a dynamically refreshed `JList`. Upon initialization, a `RentalShop` instance is constructed, loading or creating its state via the same serialization and file-locking mechanisms as the CLI. Interactive controls at the bottom of the panel let the user `RENT`, `RETURN`, `LIST`, and `TRANSACTIONS` by assembling commands, delegating to `RentalShop.executeCommand(...)`, and appending results to a `JTextArea`.

7.4 Vehicle.java and Transaction.java

Vehicle Encapsulates the immutable fields `licensePlate` and `type`, plus a mutable `kilometers` counter. Implements `Serializable` to allow binary state snapshots. Provides getters, `addKilometers(...)` for odometer updates, and a `toString()` for display.

Transaction Records a single return event with plate, kilometers, discount flag, and computed charge. Implements `Serializable`, provides getters, and overrides `toString()` to format logs in both console and human-readable dumps.

8 Output Terminal Examples

```
$ java ... RentalShop --location=SanJose --spaces-available=3 --lots=Central
```

```
Rental Shop at SanJose ready...
> RENT SUV
RENT: Obtained vehicle ABC-123 (SUV) from lot with 10% discount.
> LIST
----- Shop State (SanJose) -----
Parking Spaces Available: 2
Vehicles in Shop Inventory:
   XYZ-456 (SEDAN), Km: 0
Vehicles Rented Out:
   ABC-123 (SUV), Km: 0
Cash Earned: $0.0
> RETURN ABC-123 100
RETURN: Vehicle ABC-123 returned. Km added: 100. Charge: $90.0
> TRANSACTIONS
Vehicle ABC-123 | Km: 100 | Discount: 10% | Charge: $90.0
Total Earnings: $90.0
Total Lost Due To Discounts: $10.0
```

9 Testing Scenarios

9.1 Lot Manager Tests

- `lot-scenarios.sh` applies: `-add-sedan=2`, `-add-suv=1`, checks that `Central.txt` contains exactly three lines (two SEDAN, one SUV).
- Removal test: `-remove-vehicle=XYZ-999` should leave file unchanged and print "not found".

9.2 Rental Shop Tests

- `shop1-commands.txt`: sequence of RENT and RETURN for `SanJose`, validates output and `SanJose.txt`.
- `shop2-commands.txt`: similar for `Alajuela`, including cross-shop returns via `rented_registry.txt`.
- `rental-scenarios.sh` drives both scenarios, capturing outputs in `shop1-output.txt`, `shop2-output.txt`.

10 Extra Credit: UI Implementation

A desktop GUI was implemented in `CarRentalUI.java` using Swing. It fully supports all LotManager and RentalShop CLI commands:

- **Tabbed Interface:**
  - *Lot Manager* tab for creating lots and adding/removing sedans, SUVs, and vans.

  - *Rental Shop* tab for configuring a shop, selecting lots, and performing rentals/returns.

- **Lot Manager Panel:**
  - *Top Row:* Text field for lot name, spinners and buttons to add sedans, SUVs, and vans, plus Remove Vehicle and Refresh Lot buttons.
  - *Center:* Non-editable `JTextArea` that shows console output and the contents of the lot file.

- **Rental Shop Panel:**
  - *Configuration Row:* Text field for shop location, spinner for number of spaces, and a list of available lots with Refresh and Initialize buttons.
  - *Action Row:* Dropdown to choose vehicle type (SEDAN/SUV/VAN), text fields for license plate and kilometers, and buttons for RENT, RETURN, LIST, and TRANSACTIONS.
  - *Center:* Non-editable `JTextArea` that displays rental shop logs and results.

- **Backend Integration:** All button actions delegate to existing command-line logic:
  - Lot commands invoke `LotManager.main(...)` and capture its `System.out`.
  - Shop commands call `RentalShop.executeCommand(...)`.
  - The lots list is refreshed by scanning `*.txt` files in the working directory.

This GUI enhances usability while preserving the original CLI behavior for grading and automation.

11 AI Usage Disclosure

Per course policy, it is declared that:

- **Global-Registry Sync (AI-generated)**
  To ensure concurrency safety when multiple shop instances access the shared `rented_registry.txt`, an AI-suggested file-locking pattern was adopted. This code fragment was provided by the AI and then rigorously tested and adapted for the data model.

  Acclaimed AI-generated method: syncWithGlobalRegistryOnStartup()

  AI Prompt: "How can I ensure that different processes do not interfere, maybe by using File locking please explain it and give me an example/hint."

```
private void syncWithGlobalRegistryOnStartup() {
    Set<String> globalPlates = new HashSet<>();
    File file = new File(RENTED_REGISTRY);
    if (file.exists()) {
        try (RandomAccessFile raf = new RandomAccessFile(file, "r");
             FileChannel ch = raf.getChannel();
```

```
                 FileLock lock = ch.lock(0L, Long.MAX_VALUE, true)) {
                String line;
                while ((line = raf.readLine()) != null) {
                    globalPlates.add(line.split(",")[0]);
                }
            }
        }
        rentedVehicles.keySet()
                      .removeIf(plate -> !globalPlates.contains(plate));
    }
```

Why it works: FileChannel's `lock(..., true)` acquires a shared lock, preventing writers from interfering while reading. After loading the global plate list, any locally-rented plate not in that set is dropped, guaranteeing consistency across processes.

Why chosen: This pattern is straightforward, uses only core Java I/O, and avoids third-party dependencies.

Improvement: Before, concurrent runs could corrupt the registry or miss removals; now, all start-up syncs are safe and deterministic.

- **Swing UI Skeleton (AI-generated)**
  The initial layout and event-wiring for `CarRentalUI.java` were drafted by the AI. It was then hand-tuned the component sizes, labels, and action listeners for full coverage of CLI commands.

  AI Prompt: "Help me creating a base for a RentalShop and LotManager UI in Java. It should be basic like a good starting point. Also explain me every library and method you are using, so I can understand what is going on. I want to be able to represent the LotManager and the RentalShop classes."

  Key AI-generated code snippets include:

```
// Create main tabs and panels
JTabbedPane tabs = new JTabbedPane();
tabs.addTab("Lot Manager", buildLotManagerPanel());
tabs.addTab("Rental Shop",  buildShopPanel());
add(tabs, BorderLayout.CENTER);
pack(); setLocationRelativeTo(null); setVisible(true);
```

  and, in the Lot Manager panel:

```
lotNameField = new JTextField(10);
sedanSpinner = new JSpinner(new SpinnerNumberModel(1,1,100,1));
addSedanBtn   = new JButton("Add Sedan");
addSedanBtn.addActionListener(e -> runLotCommand("--add-sedan"));
```

Why it works: Swing's `BorderLayout` and `JTabbedPane` provide a clean separation of concerns and an intuitive user flow. Spinners ensure only valid numeric input, and centralized `runLotCommand(...)` bridges GUI actions to existing CLI logic.

Why chosen: This skeleton rapidly produced a full-feature GUI without reinventing layouts or wiring patterns.

Improvement: Contrast with a manual build—AI-seeded code saved 3–4 hours of boilerplate, allowing focus on user feedback and file-watching logic.

- **Manual Code (author-written)**
  All other functionality—flag parsing, core lot and transaction logic, human-readable state dumps, test harness integration—was designed, implemented, and verified by hand.

Integration and Validation. Every "AI-generated" fragment was:

1. Reviewed line-by-line for correctness and security.
2. Integrated into the existing architecture with appropriate error handling.
3. Tested under concurrent shop launches, lot sizes, and file-I/O edge cases.

This ensures AI assistance accelerated development without compromising reliability, maintainability, or grading reproducibility.
Comments prefaced with "AI-generated" mark the original AI output; each has been verified and, where necessary, refactored by me to meet project standards.