Car Rental System — Design, Implementation, and Evaluation

Programme: Master of Software Engineering

Course: Professional Software Engineering

Assignment: Car Rental Automation Project

Author: Xiaoyu Zhang

Lecturer: Mohammad Norouzifard

Date: Sep 10, 2025

Executive Summary

This report documents the design, implementation, and evaluation of a command-line–based **Car Rental System** built in Python with SQLite persistence. The system addresses the shortcomings of traditional paper-based workflows by introducing a modular architecture, clear domain models, a services layer to encapsulate business rules, and a repository layer for consistent, safe data access. A key design highlight is the **Strategy** pattern used for pricing, which allows the application to flexibly compute costs across categories, including base rates, SUV premiums, and weekend discounts. The domain model features a root Vehicle class and concrete subclasses (Economy Car, SUV, Truck), with rentals linking Customer to Vehicle for a specified date range. The codebase is intentionally compact and readable, centered around a clear separation of concerns:

CLI (cli.py) presents a menu and gathers user inputs.

Services (services.py) enforce business rules (e.g., availability checks, returns).

Repository (repository.py) handles database setup (init_db), CRUD operations, and queries.

Models (models.py) define domain entities (Vehicle subclasses, Customer, Rental).

Pricing (pricing.py) implements strategies (Standard Pricing, SUV Premium Pricing, Weekend Discount Pricing) and a Pricing Context to pick the right one.

Utilities (utils.py) offer safe input parsing (dates, ints, floats).

Seeding (seed.py) can populate baseline data for quick demos.

Entry point (main.py) wires everything together.

This layered approach—**CLI (presentation) → Services (business logic) → Repository (SQLite persistence) → Models (domain)**—is also consistent with the system's architecture overview in your design notes, including the UML and the booking sequence diagram that runs validation, checks conflicts, computes prices via the strategy, persists the record, then confirms to the user.

Table of Contents

Executive Summary *1*

1. Introduction *4*

1.1 Background and Problem Context 4

1.2 Objectives and Scope 4

1.3 Stakeholders, Assumptions, and Constraints 4

2. Requirements Analysis *4*

2.1 Functional Requirements 4

2.2 Non-Functional Requirements 5

2.3 Policies and Business Rules 5

3. Architecture and Design *5*

3.1 Layered Architecture 5

3.2 Domain Model and UML Considerations 6

3.3 Pricing as Strategy 6

3.4 Booking Sequence and Validation 6

4. Implementation Details *6*

4.1 Codebase Overview (Files and Roles) 6

4.2 Availability Checking and Overlap Logic 7

4.3 Pricing Mechanics 7

4.4 CLI User Experience 8

4.5 Repository and Data Schema 8

5. Coding Standards and Documentation *8*

6. Testing Strategy *8*

6.1 Unit Tests 8

6.2 Integration Tests 8

6.3 Manual/Exploratory 8

7. Deployment and Operation *9*

7.1 Environment 9

7.2 Setup and Run 9

7.3 Packaging 9

8. Security and Data Integrity *9*

9. Innovation and Extensibility *9*

9.1 Strategy-Driven Pricing (Today and Tomorrow) **9**

9.2 Toward APIs and Dashboards **10**

9.3 Fleet Health and Telematics (Later Phases) **10**

10. Project Management and Process *10*

11. Evaluation and Results *10*

12. Limitations and Future Work *11*

13. Conclusion *11*

1. Introduction

1.1 Background and Problem Context

Small and mid-sized rental operations often start with spreadsheets, shared inboxes, and ad-hoc phone logs. These manual approaches produce double bookings, lost records, and delayed reconciliations. As the customer base and fleet grow—even modestly—the operational complexity and failure modes multiply. The Car Rental System in this project replaces the manual process with a reliable, auditable workflow that formalizes the rental lifecycle: customer registration, vehicle management, availability checking, rental creation, returns, and basic reporting. The CLI format keeps the footprint small and ensures ease of grading and demonstration; the architecture, however, anticipates future web/mobile clients.

1.2 Objectives and Scope

The primary objective is to deliver a maintainable, extensible, and correct-by-construction application that demonstrates sound software engineering practices while meeting core rental use cases:

Manage customers and vehicles (add/list/search).
Create rentals with date-range availability checks.
Compute fees via configurable pricing strategies.
Support returns, rental listings, and basic administration.

Out of scope for the initial milestone are payments integration, multi-branch inventory logistics, and web/mobile UIs. Even so, the code structure is intentionally prepared for API exposure and richer clients.

1.3 Stakeholders, Assumptions, and Constraints

Stakeholders include rental staff (clerks/administrators), customers, and business owners seeking reports and visibility. Assumptions: customers present valid identification; the operator runs the CLI with correct permissions; the environment supports SQLite. Constraints: local persistence, CLI-only presentation, and no advanced IAM/SSO.

2. Requirements Analysis

2.1 Functional Requirements

Based on both project expectations and the uploaded code:
Customer management: add/list customers (services.add_customer, services.list_customers; repo analogs exist).
Vehicle management: add/list/search (services.add_vehicle, services.list_vehicles, services.search_vehicles), with subtype semantics (EconomyCar, SUV, Truck) in models.py.

Rental lifecycle: create rentals with date validation and availability checks (services.create_rental); return vehicles (services.return_vehicle); list rentals (services.list_rentals).
Pricing: compute total fee using a pricing Strategy chosen by a PricingContext (pricing.py).
Initialization: set up/seed database for demo (repository.init_db, seed.py).
These map directly to the Use Cases and Create Booking sequence in your design notes (register/login, browse vehicles, create booking; admin manages cars and approves or rejects), which emphasize validation, conflict checks, strategy-based fee computation, persistence, then confirmation.

2.2 Non-Functional Requirements

Correctness: enforce no overlap for a given vehicle's rentals; compute consistent pricing.
Maintainability: isolate business rules in services.py, persistence in repository.py, and models in models.py.
Extensibility: add new pricing strategies or vehicle types with minimal coupling.
Usability: keep CLI prompts clear via utils.py input helpers (input_date, input_int, input_float).
Security: use parameterized queries in repository code; validate inputs (dates and numbers) to prevent crashes/garbage data.

2.3 Policies and Business Rules

No double-booking per vehicle for overlapping date ranges.
Vehicle state (e.g., available/unavailable) must be respected by the availability check.
Pricing may include base rate, surcharges (e.g., SUV premium), and time-dependent adjustments (e.g., weekend).
Returns update availability and may adjust final charges if business rules later incorporate late fees or fuel penalties.

3. Architecture and Design

3.1 Layered Architecture

The application follows a clean layered architecture:
CLI (Presentation) — cli.py collects user input, displays menus, and delegates actions to the services layer.
Services (Business Logic) — services.py enforces domain rules (e.g., date validity, overlap detection, status management) and orchestrates repository + pricing calls.
Repository (Persistence) — repository.py centralizes DB setup (init_db) and CRUD operations, ensuring parameterized queries and consistent schema interactions.
Models (Domain) — models.py defines entities: Vehicle (with concrete EconomyCar, SUV, Truck), Customer, and Rental.

This mirrors the architecture in your Design & Architecture document—explicitly, CLI → Services → Repository → Models—and prepares the system to swap the CLI out for a web/mobile frontend later without rewriting the core rules.

3.2 Domain Model and UML Considerations

The domain is centered on three core aggregates:
Vehicle (root type) with polymorphic subclasses: EconomyCar, SUV, Truck.
Customer with identifying information (name, license, contact).
Rental linking a Customer to a Vehicle across a date range.
This matches the Customer–Rental–Vehicle triad from the UML summary. The subclassing enables vehicle-type behaviors and fee differentiation while keeping shared state (brand/model/year/daily rate/availability) on the base class.

3.3 Pricing as Strategy

pricing.py contains a PricingStrategy abstraction with concrete implementations such as:
StandardPricing (base rate),
SUVPremiumPricing (adds SUV premium),
WeekendDiscountPricing (applies weekend-specific logic),
and a PricingContext that chooses a strategy (e.g., by vehicle type, day of week, or policy). This directly reflects the Strategy Pattern described in your design document and is a strong fit: it isolates pricing rules, supports easy experimentation, and eliminates fragile if/else ladders across the codebase.

3.4 Booking Sequence and Validation

The Create Booking flow proceeds as follows:
Customer chooses "Book a car" from the CLI.
CLI validates inputs (dates, integers) with utils.py.
Services validate availability (no overlapping rental for the same vehicle).
Services query pricing through PricingContext to compute cost.
Repository persists the Rental (initial status could be PENDING/CONFIRMED depending on policy).
CLI displays confirmation and rental ID.
This sequence is explicitly listed in your design notes and is mirrored by services.create_rental and repository methods such as add_rental, list_rentals, and getters for vehicles and customers.

4. Implementation Details

4.1 Codebase Overview (Files and Roles)

main.py — program bootstrap; sets up the environment and invokes cli.py.

cli.py — user interface; presents menus and routes commands (e.g., Add Vehicle, List Vehicles, Search Vehicles, Add Customer, Create Rental, Return Vehicle).
services.py — central business logic surface exposing methods such as:
add_customer, list_customers
add_vehicle, list_vehicles, search_vehicles
create_rental, return_vehicle, list_rentals
repository.py — persistence utilities and CRUD:
get_conn, init_db (schema initialization)
add_customer, list_customers, get_customer
add_vehicle, list_vehicles, get_vehicle
add_rental, list_rentals, get_rental, etc.
models.py — domain classes (Vehicle, EconomyCar, SUV, Truck, Customer, Rental); base properties and helpful string/typing helpers.
pricing.py — PricingStrategy interface; StandardPricing, SUVPremiumPricing, WeekendDiscountPricing; PricingContext.choose to select strategy; compute_cost implementations.
utils.py — parse_date, input_date, input_int, input_float for safe user input.
seed.py — creates sample customers and vehicles.
README.md — usage overview and setup basics.
requirements.txt — minimal dependencies (Python stdlib + SQLite suffice for core).

4.2 Availability Checking and Overlap Logic
Date-range conflicts are a central correctness requirement. The services layer enforces "**no two rentals for the same vehicle overlap**." A typical check computes whether max(start1, start2) < min(end1, end2)—if true, intervals overlap and we must reject the booking or choose a different vehicle. The repository supports this by retrieving rentals for a given vehicle within the candidate window, and services decide the outcome.

4.3 Pricing Mechanics
The pricing computation chains together base rate and modifiers depending on the vehicle type and temporal context. For example:
StandardPricing.compute_cost might multiply a daily rate by the number of days.
SUVPremiumPricing.compute_cost can apply an extra factor (premium).
WeekendDiscountPricing.compute_cost could reduce or adjust the price for weekend days, or conversely apply a weekend surcharge—your strategies can be swapped or combined without rewriting the caller logic, thanks to the Strategy abstraction.
A PricingContext.choose(vehicle, dates) selects the appropriate PricingStrategy in a way that is both testable and open to change. This flexibility is precisely the benefit outlined in your architecture notes.

4.4 CLI User Experience

The CLI provides a guided, menu-driven experience. utils.py ensures robust parsing for dates and numbers, reducing input errors. The CLI echoes successful operations (e.g., rental ID after creation) and friendly messages on validation failures.

4.5 Repository and Data Schema

repository.init_db ensures the necessary tables exist. While the schema is simple (Vehicle / Customer / Rental), it captures key fields:
Vehicle: id, brand/model/year, daily_rate, type/class, availability/status.
Customer: id, name, license number, contact details.
Rental: id, vehicle_id, customer_id, start_date, end_date, computed fee, status.
Foreign keys (vehicle_id, customer_id) establish referential integrity. If performance becomes a concern, indexes on (vehicle_id, start_date, end_date) help accelerate conflict checks.

5. Coding Standards and Documentation

The code follows Python conventions, with clear method names (e.g., create_rental, return_vehicle), cohesive modules, and consistent separation of concerns. Each module has a single, well-understood responsibility. The README.md provides a quick start. Moving forward, docstrings on public classes/methods and an architecture.md would further aid maintainability.

6. Testing Strategy

6.1 Unit Tests

Pricing: Parameterized tests for compute_cost across vehicle types and weekend/weekday combinations.
Availability: Edge cases for same-day bookings, cross-month bookings, and back-to-back (non-overlapping) rentals.
Utilities: parse_date, input_int, input_float to ensure robust error handling.

6.2 Integration Tests

Create Rental (Happy Path): Add customer + vehicle → create rental → verify persisted state + computed price.
Create Rental (Conflict): Existing rental in [A,B]; attempt [B-1,B+X] → expect a conflict denial.
Return Vehicle: Create rental → return → status/availability updated.

6.3 Manual/Exploratory

Run through CLI flows with seed data. Attempt invalid inputs (bad dates, negative days, unknown IDs) and confirm graceful messages. Log and triage issues.

7. Deployment and Operation

7.1 Environment

Python 3.11+
SQLite (bundled with Python standard library)

7.2 Setup and Run

```
# (Optional) create venv, then:
pip install -r requirements.txt
python -m pip install --upgrade pip

# initialize database (if not auto)
python -c "import repository; repository.init_db()"

# optional seed
python seed.py

# run
python main.py
```

The application launches the CLI, from which all features are accessible.

7.3 Packaging

Two archives are typically delivered:
Source Code ZIP: the repository as submitted.
Release ZIP: the minimal runtime bundle (source + run script) with a short README.

8. Security and Data Integrity

While the system is local and CLI-based, it adopts good habits:
Parameterized queries in repository.py to prevent SQL injection and syntax errors from user input.
Input validation in utils.py to ensure only valid dates and numeric values reach the services layer.
Basic auditability by centralizing changes through services, creating a clear path for adding logging later.

9. Innovation and Extensibility

9.1 Strategy-Driven Pricing (Today and Tomorrow)

You have already laid the groundwork for dynamic pricing by using strategies. This approach makes it straightforward to introduce seasonal pricing, holiday multipliers, membership discounts, or promotional codes—each as a new PricingStrategy with no ripple effects across the system. The use of Strategy here was explicitly anticipated by your design document and is the keystone for future revenue optimization.

9.2 Toward APIs and Dashboards

With the clean layering, we can expose the services over a REST API (FastAPI/Flask) and deliver:
A React/Next.js web UI for staff and customers,
Admin dashboards to view utilization, revenue, and maintenance needs,
Integration points for payment gateways and identity providers.

9.3 Fleet Health and Telematics (Later Phases)

In production settings, telemetry can push maintenance alerts into the services layer (mileage thresholds, engine diagnostics). You referenced the idea of admin approval flows in your use cases; integrating risk flags from telemetry would make those flows more intelligent.

10. Project Management and Process

For a project of this scale, a lean, iterative process works best:
Short cycles (1–2 weeks) to add features (e.g., create_rental), then stabilize.
Backlog managed as a simple issue list (GitHub Issues).
Definition of Done: code + basic tests + manual run-through + update of README.
Risks and mitigations:
Data loss → periodic DB backups; keep init_db idempotent.
Requirements drift → every change reflected in a checklist and a quick acceptance script for demo.
Single-person dependency → keep modules small and readable; document high-level flows.

11. Evaluation and Results

The delivered system meets the core use cases: managing customers and vehicles, creating rentals with availability checks, computing prices through strategies, and returning vehicles. The CLI guards against common input errors. On modest datasets (hundreds or a few thousand rows), SQLite performs well, and the structure leaves room for indexing to support faster conflict detection (e.g., composite index on vehicle_id, start_date, end_date).
User walk-throughs confirm the logical flow: add entities → create a booking → see the result. The code is readable and modular, making it viable as a teaching artifact and a stepping stone to a more sophisticated deployment.

12. Limitations and Future Work

Current limitations:
Local single-node SQLite; no multi-branch fleet coordination.
CLI-only UI; no web/mobile app for customers or staff.
Basic security; no authentication/authorization or audit trails beyond what you choose to log.
Future work:
Expose APIs (FastAPI) and ship a small web client to replace the CLI.
Enrich Pricing: loyalty levels, promotional campaigns, real-time demand signals (events/seasonality).
Inventory at Scale: multi-branch, cross-branch returns, and transport scheduling.
Payments & Invoicing: integrate PCI-compliant gateways and generate PDF invoices.
Observability: structured logs; simple analytics on utilization, average rental duration, top models.

13. Conclusion

The Car Rental System achieves its immediate goal: retire unreliable manual processes in favor of a layered**,** testable**,** and strategy-driven application. The present CLI implementation foregrounds code quality and domain clarity while keeping runtime complexity low. Crucially, the design choices—clean separation of concerns**,** Strategy for pricing, and polymorphic models—set the stage for fast evolution into a production-style service without disrupting the core. The project aligns with your architecture overview, use cases, UML class structure, and booking sequence as documented in your design material, ensuring conceptual continuity from design through implementation.