2026

Parking Lot Management System

SOFTWARE DESIGN & ARCHITECTURE PROJECT

BY ELLA OBIORA

Objective

The Goal of the redesign was to improve the maintainability and correctness of the original Parking Lot Manager by:

- Introducing **two object-oriented design patterns** (**Factory** and **Command**)
- Removing **high-impact anti-patterns** and fixing functional defects
- Improving separation of concerns between **UI (Tkinter)** and **domain logic (ParkingLot)**

This redesign preserves the original user-facing behavior (GUI workflow remains the same) while restructuring the internal architecture to be cleaner and more extensible.

Work Completed

- Created Original UML artifacts: structural (class diagram) + behavioral (sequence diagram).
- Created Redesign UML artifacts reflecting Factory + Command refactor and structural fixes.
- Refactored codebase into a UI layer (Tkinter) and a domain layer (ParkingLot + vehicle models).
- Re-validated application behavior via GUI run-proof screenshots after refactor.

Submission Package Checklist

- Original UML — class diagram (structural)
- Original UML — sequence diagram (behavioral)
- Redesign UML — class diagram (structural)
- Redesign UML — sequence diagram (behavioral)
- Updated source code (ParkingManager.py, Vehicle.py, ElectricVehicle.py, and any new pattern classes)
- Run-proof screenshots (GUI running + key flows: create lot, park regular, park EV, status, charge status, remove)
- Written design justification / DDD + microservices section (bounded context diagram + basic domain models + high-level architecture)

Original Design Issues (Anti-Patterns / Defects Identified)

A) Tight UI Coupling (God Object / Mixed Responsibilities)

In the original implementation, ParkingLot (domain logic) directly:

- Read UI inputs (StringVar, IntVar)
- Wrote UI output to tfield (Tkinter text box)

This mixed UI and business logic into the same class, making the domain model difficult to test, reuse, or refactor.
Fix: ParkingLot no longer reads or writes UI elements. The UI layer is responsible for display and input handling.

B) Slot Indexing Bug / Inconsistent Slot IDs

Originally, park() used internal counters (slotid, slotEvId) that did not match actual array positions after cars were removed. This could lead to incorrect slot identification over time.

Fix: Slot assignment is now based on the **actual empty index** in the list:

- Internal storage uses list indices (0-based)
- Returned slot number is always **index + 1** (user-facing 1-based)
- leave() consistently converts slotid to index via slotid - 1

This makes slot behavior stable even after removals.

C) EV Helper Methods Bug (Undefined Variables)

Original EV lookup methods referenced variables that didn't exist (ex: make, model inside functions that took color).

Fix: Updated method signatures and logic:

- getSlotNumFromMakeEv(self, make)
- getSlotNumFromModelEv(self, model)

Now they correctly filter EV slots using the passed-in argument.

D) Incorrect Inheritance for ElectricCar / ElectricBike

Originally, ElectricCar and ElectricBike called ElectricVehicle.__init__() but did not inherit from ElectricVehicle, which is confusing and breaks proper OO design.

Fix: Made them true subclasses:

- class ElectricCar(ElectricVehicle)
- class ElectricBike(ElectricVehicle)
  and used super().__init__(...)

This ensures consistent type behavior and cleaner polymorphism.

Design Patterns Implemented

Pattern #1 — Factory Pattern (VehicleFactory)

VehicleFactory centralizes creation of Car/Motorcycle and ElectricCar/ElectricBike objects.

Problem in original:

- Vehicle creation logic was embedded directly inside ParkingLot.park() with branching rules for:
- EV vs non-EV
- Motorcycle vs Car

  This increases complexity and makes it hard to extend with new vehicle types.

Solution:

- Implemented a centralized factory:
- **VehicleFactory.create_vehicle(regnum, make, model, color, ev, motor)**
- This factory encapsulates all object creation rules.

  ParkingLot now focuses only on domain rules (finding empty slots, storing vehicles), not instantiation details.

Benefits:

- Eliminates duplication and branching in business logic
- Easier extension (new vehicle types can be added in one place)
- Cleaner separation of responsibilities

Pattern #2 — Command Pattern (UI Button Actions)

Each UI button is wired to a dedicated Command class that calls domain methods and writes outputs.

Problem in original:

- UI buttons called methods inside ParkingLot that also contained UI dependencies, causing tight coupling and poor modularity.

Solution:

- Each UI action is represented by its own command class:
  - CreateLotCommand
  - ParkCarCommand
  - RemoveCarCommand
  - GetSlotByRegCommand
  - GetSlotByColorCommand
  - GetRegByColorCommand
  - StatusCommand
  - ChargeStatusCommand

- Each command object:
  - Reads UI inputs (StringVar/IntVar)
  - Calls **pure domain methods** in ParkingLot
  - Writes outputs to tfield

Benefits:

- Strong separation between **UI event handling** and **domain model**
- Clear mapping between GUI buttons and program behavior
- Easier debugging and easier future changes (add/modify commands without touching domain logic)

DDD + Microservices Design (EV Charging Extension)

Ubiquitous Language

- ParkingLot: a parking facility on a specific floor level with regular and EV slots.
- Slot: a numbered position (regular slot or EV slot).
- Vehicle: a car or motorcycle identified by RegistrationNumber and described by make/model/color.
- EV: an electric vehicle (ElectricCar or ElectricBike) with a ChargeLevel.
- ChargingStation: equipment that can charge EVs.
- ChargingSession: a time-bound session that tracks charging progress for a specific EV.

Bounded Contexts

- Parking Management Context: allocating slots, releasing slots, and searching vehicles/slots.
- EV Charging Context: tracking charge level, charging sessions, and station availability.

A bounded context diagram is included in the submission to show these contexts and their relationship (integration via events or API calls).

Domain Models (Entities, Value Objects, Aggregates)

Parking Management Context:

- Aggregate: ParkingLot (root) - owns Slot entities and enforces slot allocation rules.
- Entities: Slot, Vehicle (Car/Motorcycle).
- Value Objects: RegistrationNumber, Color, Make, Model, FloorLevel.

EV Charging Context:

- Aggregate: ChargingStation (root) - owns ChargingSession entities and manages charging availability/capacity.
- Entities: ChargingSession, ElectricVehicle (ElectricCar/ElectricBike).
- Value Objects: ChargeLevel (0-100), StationId, SessionId, Timestamp..

Domain Events

- VehicleParked(registrationNumber, slotNumber, isEV).
- VehicleRemoved(registrationNumber, slotNumber, isEV).
- ChargingStarted(registrationNumber, stationId, sessionId).
- ChargeUpdated(registrationNumber, sessionId, chargeLevel).
- ChargingCompleted(registrationNumber, sessionId).

Proposed Microservices Architecture

Services (each with its own database):

- Parking Service: manages parking lots, slot allocation, and vehicle lookup (DB: ParkingDB).
- EV Charging Service: manages charging stations, sessions, and charge status (DB: ChargingDB).

A high-level microservices architecture diagram is included in the submission. It shows service boundaries, request flows, and separate databases per service, aligned to the bounded contexts.

Structural Fixes Applied

- Removed tight coupling between ParkingLot and Tkinter state (StringVar/Text widget): domain returns values/text; UI renders.
- Standardized slot indexing: internal arrays remain 0-based; UI-visible slot IDs are 1-based and consistently converted.
- Corrected EV helper methods to use defined parameters (make/model) and consistent EV slot iteration.
- Converted ElectricCar and ElectricBike into true subclasses of ElectricVehicle (inheritance + super()).

Structural Improvements in Redesign

A) ParkingLot is now a clean domain model

ParkingLot now contains only:

- Parking rules
- Slot allocation logic
- Queries (lookups)
- Formatting helpers (status_text(), charge_status_text())
- It does **not** touch Tkinter UI objects.

B) Stable Slot Management

Slot management is consistent:

- Stores vehicles in a list (slots and evSlots)
- Uses the first available empty index
- Returns 1-based slot numbers to match the UI
- Removes vehicles using validated bounds checking

C) Improved Extensibility

The new design supports future upgrades without major changes:

- Add a new vehicle type → update factory only
- Add a new UI feature/button → create a new command class
- Add new domain rules → modify ParkingLot without UI impact

Summary of "Before vs After"

Before

- Domain logic and UI logic were mixed
- Slot assignment could break after removals
- EV lookup methods contained variable errors
- Electric vehicle classes didn't inherit properly
- Object creation scattered across business logic

After

- Clear separation: UI (Commands) vs Domain (ParkingLot)
- Slot numbering and removal logic is consistent and reliable
- EV helper methods corrected and functional
- ElectricCar and ElectricBike are true subclasses of ElectricVehicle
- Vehicle creation centralized via Factory pattern
- GUI behavior preserved, architecture improved

How to Run the Application

From the project folder that contains ParkingManager.py, run:

```
cd .\parking_project
python .\ParkingManager.py
```

Conclusion

The redesigned implementation improves architecture by applying the Factory and Command patterns, removing major coupling issues between UI and domain logic, fixing slot ID correctness, and correcting EV model inheritance and helper methods. The included DDD and microservices proposal demonstrates how the system can be evolved into bounded contexts with clear service boundaries and separate databases per service, meeting the rubric criteria for a high-quality design submission.