# Domain Driven Design for *EasyParkPlus*

## Software Design & Architecture Project

---

October, 2025

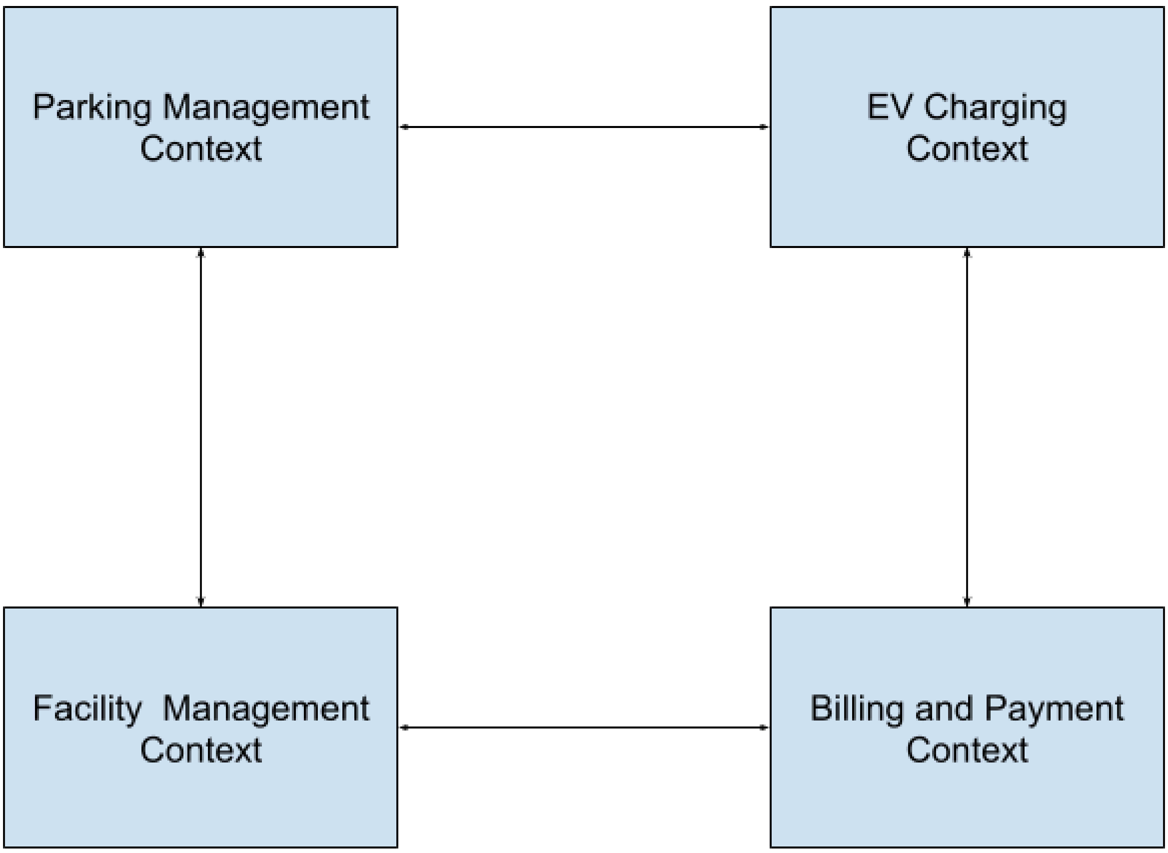
# Table of Contents

This is a preliminary microservices architecture that will help to identify services within the bounded contexts that will be described, key responsibilities of each of the services, describe API endpoints for the external-facing and service-to-service endpoints and finally, the identification of separate DBs per service.

This document explains the Domain-Driven Design approach, microservices architecture, and design patterns applied to the EasyParkPlus, focusing on both core parking management and EV charging capabilities.

# Domain Analysis and Bounded Contexts

## High-Level Bounded Context Diagram



# Domain Models

---

## 1. Parking Management Context

**Core Entities**

- ParkingFacility
  - Attributes: ID, name, location, total spaces
  - Responsibilities: Facility-level operations
- ParkingSpace
  - Attributes: ID, status, type (regular, handicap, EV)
  - Responsibilities: Space allocation and status tracking
- Vehicle
  - Attributes: license plate, type, size
  - Responsibilities: Vehicle information management
- ParkingSession
  - Attributes: start time, end time, space ID, vehicle ID
  - Responsibilities: Track parking duration and usage

**Value Objects**

- Location: Geographic coordinates
- SpaceType: Enumeration of space types
- VehicleType: Enumeration of vehicle categories

## 2. EV Charging Context

**Core Entities**

- ChargingStation
  - Attributes: ID, status, power rating, connector types
  - Responsibilities: Charging station management
- ChargingSession
  - Attributes: start time, end time, energy consumed, vehicle ID
  - Responsibilities: Track charging sessions

**Value Objects**

- PowerRating: Charging capacity details
- ConnectorType: Available connector standards
- ChargingStatus: Current station status

# 3. Facility Management Context

### Core Entities

- Facility
  - Attributes: ID, name, address, operating hours
  - Responsibilities: Overall facility management
- MaintenanceSchedule
  - Attributes: facility ID, maintenance type, schedule
  - Responsibilities: Track facility maintenance

### Value Objects

- Operating Hours: Facility timing details
- MaintenanceType: Types of maintenance activities

# 4. Billing & Payment Context

### Core Entities

- Bill
  - Attributes: ID, amount, services used, status
  - Responsibilities: Payment processing
- PaymentTransaction
  - Attributes: transaction ID, payment method, amount
  - Responsibilities: Handle payment operations

### Value Objects

- Money: Amount and currency
- PaymentStatus: Current payment state

# Microservices Architecture

---

## 1. Parking Service

- **Responsibilities**
  - Manage parking spaces and vehicle entry/exit
  - Track parking sessions
  - Handle space allocation logic
- **APIs/Endpoints**

```
External:
POST   /api/parking/spaces          # Get available spaces
POST   /api/parking/vehicle/park    # Park a vehicle
DELETE /api/parking/vehicle/{id}    # Remove vehicle
GET    /api/parking/status          # Get lot status

Internal:
GET    /internal/parking/space/{id} # Get space details
POST   /internal/parking/validate   # Validate parking session
```

- **Database**
  - ParkingDB (PostgreSQL)
    - Tables: spaces, vehicles, parking_sessions
    - Optimized for real-time space management

## 2. EV Charging Service

- **Responsibilities**
  - Manage charging stations
  - Control charging sessions
  - Monitor power consumption

**APIs/Endpoints**

```
External:
POST   /api/charging/start          # Start charging
POST   /api/charging/stop           # Stop charging
GET    /api/charging/stations       # List stations
GET    /api/charging/status/{id}    # Station status

Internal:
POST   /internal/charging/validate  # Validate charging session
GET    /internal/charging/metrics   # Get power metrics
```

**Database**

- ChargingDB (MongoDB)
  - Collections: stations, charging_sessions, power_metrics
  - Optimized for time-series data

## 3. Facility Management Service

- **Responsibilities**
  - Overall facility operations
  - Maintenance scheduling
  - Resource allocation

  **APIs/Endpoints**
```
External:
GET    /api/facility/info           # Facility information
POST   /api/facility/maintenance    # Schedule maintenance

Internal:
GET    /internal/facility/status    # Operational status
POST   /internal/facility/metrics   # Update metrics
```

- **Database**
  - FacilityDB (PostgreSQL)
    - Tables: facilities, maintenance_schedules, operations
    - Optimized for operational data

## 4. Billing Service

- **Responsibilities**
  - Calculate charges
  - Process payments
  - Generate invoices

  **APIs/Endpoints**
```
External:
POST   /api/billing/calculate       # Calculate charges
POST   /api/billing/pay            # Process payment
GET    /api/billing/invoice/{id}    # Get invoice

Internal:
POST   /internal/billing/validate   # Validate payment
GET    /internal/billing/rates      # Get current rates
```

- **Database**
  - BillingDB (PostgreSQL)
    - Tables: bills, payments, transactions
    - Optimized for financial transactions

# Inter-Service Communication

---

- **Synchronous Communication**
  - REST APIs for real-time operations
  - gRPC for high-performance internal communication
- **Asynchronous Communication**
  - Message queue (RabbitMQ) for event-driven updates
  - Event bus for cross-service notifications

# Data Consistency Strategy

- **Saga Pattern** for distributed transactions
- **Event Sourcing** for audit trails
- **CQRS** for complex queries without impacting transaction performance

# Security Considerations

- JWT-based authentication
- Service-to-service API keys
- Role-based access control (RBAC)
- API Gateway for external request handling

# Scalability Approach

- Horizontal scaling of services
- Database sharding for large datasets
- Caching layer (Redis) for frequent queries
- Load balancing across service instances

2. Removed anti-patterns and fixed issues:
   - Eliminated module-level GUI globals and encapsulated the GUI into `ParkingApp`.
   - Fixed incorrect or non-idiomatic inheritance in `ElectricVehicle.py`.
   - Replaced inconsistent method names with a consistent snake_case API and added backward-compatible camelCase delegators where appropriate.
   - Removed dead/duplicate GUI code and copy-paste bugs (e.g., wrong parameter names in EV query methods).
   - Replaced implicit assumptions about global state with explicit dependencies (GUI renders what the business logic returns).
3. Improved code quality:
   - Added docstrings, light typing hints, and in-code comments describing design decisions.
   - Centralized object creation (via Factory) to simplify extension and reduce duplication.
   - Separated responsibilities: `ParkingLot` contains business logic; `ParkingApp` owns UI concerns.

## Why these patterns

Factory (VehicleFactory)

- Problem addressed: `ParkingLot` previously instantiated concrete classes (`Vehicle.Car`, `ElectricVehicle.ElectricCar`) directly in several places. This duplicates the decision logic and couples the lot to concrete implementations.
- Benefit: Centralizes creation in one place. When adding new vehicle types or changing constructors, only `VehicleFactory` needs change. Unit testing and mocking object creation become easier.
- Justification: *The project already contains multiple concrete vehicle types; a factory avoids spreading instantiation logic across methods (reducing repetition and single-responsibility violations)*.

Strategy (ChargingStrategy)

- Problem addressed: Charging behaviour is a concept that may change (different charge algorithms, clamping rules, scheduled charging), and the EV classes should not hard-code these policies.
- Benefit: Encapsulates charging policy behind an interface; different strategies can be injected or swapped at runtime without modifying EV or ParkingLot classes.
- Justification: Even a simple `SimpleChargeStrategy` future-proofs the codebase and demonstrates clear separation of algorithm (charging) from data structures (EV objects).

## Anti-patterns removed and why

- Global mutable GUI state: Previously `tk.StringVar()` and widgets were module-level globals. Globals make reasoning, testing, and reuse hard. Moving to `ParkingApp` makes state local and explicitly passed.
- Incorrect inheritance and direct base **init** calls: `ElectricCar`/`ElectricBike` originally called `ElectricVehicle.__init__` instead of using `super()`. This is error-prone and non-idiomatic. Fixed by using proper subclassing.
- Inconsistent naming (camelCase vs snake_case): Mixing styles reduces readability and increases cognitive load. Standardized on snake_case and preserved legacy methods where practical.
- Business and GUI coupling: `ParkingLot` previously wrote directly to a `Text` widget. This couples the business layer to a specific UI. Now `ParkingLot` returns strings and the GUI writes them to widgets.
- Copy-paste bugs and dead code: Fixed places where function parameters or local variable names were mismatched.

## Trade-offs and rationale

- Backwards compatibility vs clean API: converted getters to snake_case but left camelCase methods in `Vehicle` delegating to the new methods. This keeps external code running while moving toward idiomatic APIs.

- Simplicity vs completeness: introduced minimal, clear implementations of Factory and Strategy patterns. The goal is to show meaningful architectural improvements without making the codebase heavy.