Domain-Driven Design and Microservices Architecture Analysis

Parking Management System with EV Charging Extension

Executive Summary

This document outlines the Domain-Driven Design approach and microservices architecture for the Parking Management System, demonstrating systematic application of software engineering principles to transform a legacy codebase into a scalable, enterprise-ready solution.

---

1. Domain-Driven Design Approach

1.1 Strategic Design - Bounded Contexts

The system was analyzed through a Domain-Driven Design lens, identifying two core bounded contexts with distinct ubiquitous languages and domain models.

Bounded Contexts Identified:

🅿 Parking Operations Context
Ubiquitous Language:

- **ParkingLot** (Aggregate Root) - Physical parking facility with multiple levels
- **ParkingLot** - Individual space for vehicle parking
- **SlotAllocation** - Assignment of vehicle to specific slot
- **CheckIn/CheckOut** - Vehicle entry and exit events
- **Occupancy** - Current utilization state

Domain Entities:

- ParkingLot (Root)
- ParkingSlots (Entity)
- Floor (Value Object)
- Capacity (Value Object)

Domain Services:

- SlotAssignmentService
- OccupancyCalculator
- PricingStrategyService

⚡ EV Charging Context

Ubiquitous Language:

- **ChargingStation** - Dedicated EV charging infrastructure
- **ChargingPort** - Individual charging connector
- **ChargeSession** - Active charging period
- **EnergyDelivery** - kWh transferred
- **ConnectorType** - Physical interface standard

Domain Entities:

- ChargingStation (Root)
- ChargingPorts (Entity)
- PowerCapacity (Value Object)
- AvailabilitySchedule (Value Object)

Domain Events:

- ChargingStarted
- ChargingCompleted
- ChargingInterrupted

1.2 Context Mapping & Integration

```
[Parking Operations] ───────[Customer/Supplier]──────▶ [EV Charging]
       │                          │
       │                          │
       ▼                          ▼
[Shared Kernel] ─────────────────[Vehicle, Time, Location]─────────┘
       │
       │
       ▼
[User Management] ─────────[Conformist]────────▶ [Payment Processing]
```

1.3 Strategic Design Decisions

Aggregate Boundaries:

- **ParkingLot** protects slot allocation consistency
- **ChargingStation** manages port availability and energy allocation
- Separate aggregates prevent transactional overlap

Domain Services:

- Cross-cutting concerns handled by dedicated services
- Complex business rules encapsulated in domain services
- Stateless operations that don't fit entities/value objects

Anti-Corruption Layer:

- External systems (payment gateways) isolated
- Domain models protected from external changes
- Adapter patterns for integration

2. Domain Models

2.1 Parking Management Domain Model

python

```
# Aggregate Root
class ParkingLot:
    def __init__(self, lot_id: str, name: str, location: Location):
        self.lot_id = lot_id
        self.name = name
        self.location = location
        self.floors: List[Floor] = []
        self.pricing_strategy: PricingStrategy

    def assign_vehicle_slot(self, vehicle: Vehicle) -> SlotAssignment:
        # Domain logic for optimal slot assignment
        pass

    def calculate_occupancy_rate(self) -> Occupancy:
        # Domain logic for business intelligence
        pass

# Value Objects (Immutable)
@dataclass(frozen=True)
class Location:
    address: str
    coordinates: tuple
    timezone: str

@dataclass(frozen=True)
class PricingStrategy:
    hourly_rate: Decimal
    daily_max: Decimal
```

```
    ev_surcharge: Decimal
    motorcycle_discount: Decimal
```

2.2 EV Charging Domain Model Extension

python

```
# Aggregate Root
class ChargingStation:
    def __init__(self, station_id: str, capacity_kw: int):
        self.station_id = station_id
        self.capacity_kw = capacity_kw
        self.ports: List[ChargingPort] = []
        self.maintenance_schedule: MaintenanceSchedule

    def start_charging_session(self, vehicle: ElectricVehicle, port_id: str) ->
ChargeSession:
        # Domain logic ensuring business rules
        pass

    def calculate_energy_cost(self, session: ChargeSession) -> EnergyCost:
        # Complex pricing domain logic
        pass

# Domain Events
@dataclass
class ChargingStarted:
    session_id: str
    vehicle_id: str
    station_id: str
    start_time: datetime
    initial_charge: int
```

Enhanced EV Charging Ubiquitous Language

Core Concepts:

- **SmartCharging** - AI-optimized charging based on grid demand
- **LoadBalancing** - Distributed power allocation across ports
- **ReservationWindow -** Pre-booked charging time slots
- **PowerSharing** - Dynamic power distribution between active sessions
- **BillingTier** - Progressive pricing based on energy consumption

Technical Terms:

- **OCPP** (Open Charge Point Protocol) - Standard charging communication
- **ISO 15118** - Vehicle-to-Grid communication standard
- **DynamicLoadManagement** - Real-time power optimization
- **SessionAuthentication** - Secure charging initiation

3. Microservices Architecture Proposal

3.1 Service Decomposition Strategy

The monolithic application was decomposed into four cohesive microservices, each with specific responsibilities and bounded contexts.

3.2 Microservices Specification

🅿 Parking Service

Responsibilities:

- Slot allocation and management
- Occupancy tracking and optimization
- Check-in/check-out operations
- Multi-level parking coordination
- Real-time availability updates

API Endpoints:

http

```
POST   /api/v1/parking-lots                 # Create parking lot
GET    /api/v1/parking-lots/{id}/slots       # Get available slots
POST   /api/v1/vehicles/{id}/check-in        # Vehicle check-in
POST   /api/v1/vehicles/{id}/check-out       # Vehicle check-out
GET    /api/v1/parking-lots/{id}/status      # Current occupancy
PUT    /api/v1/parking-lots/{id}/maintenance  # Set maintenance mode
```

Database Schema (Parking DB):

sql

```
CREATE TABLE parking_lots (
   lot_id UUID PRIMARY KEY,
   name VARCHAR(100) NOT NULL,
   location JSONB,
   total_capacity INTEGER,
   current_occupancy INTEGER DEFAULT 0,
   created_at TIMESTAMP DEFAULT NOW());
```

```sql
CREATE TABLE parking_slots (
    slot_id UUID PRIMARY KEY,
    lot_id UUID NOT NULL,
    slot_number INTEGER,
    slot_type VARCHAR(20) CHECK (slot_type IN ('REGULAR', 'EV', 'MOTORCYCLE')),
    is_occupied BOOLEAN DEFAULT false,
    is_maintenance BOOLEAN DEFAULT false
);
```

⚡ Charging Service

Responsibilities:

- EV charging station management
- Charging session lifecycle control
- Energy consumption tracking and billing
- Port availability and power management
- Charging optimization algorithms

API Endpoints:

```http
POST   /api/v1/charging/stations              # Register charging station
GET    /api/v1/charging/stations/available    # Get available charging ports
POST   /api/v1/charging/sessions/start        # Start charging session
POST   /api/v1/charging/sessions/{id}/stop    # Stop charging session
GET    /api/v1/charging/sessions/{id}         # Get session status
GET    /api/v1/charging/stations/{id}/stats   # Station statistics
```

Enhanced EV Charging APIs:

```http
POST   /api/v2/charging/stations              # Register smart charging station
POST   /api/v2/charging/sessions              # Start smart charging session
POST   /api/v2/charging/reservations          # Reserve charging time slot
POST   /api/v2/charging/optimize              # Optimize charging schedule
POST   /api/v2/charging/load-balancing        # Dynamic load management
```

Database Schema (Charging DB):

sql

```
CREATE TABLE charging_stations (
    station_id UUID PRIMARY KEY,
    lot_id UUID NOT NULL,
    station_name VARCHAR(100),
    total_ports INTEGER NOT NULL,
    available_ports INTEGER DEFAULT 0,
    power_capacity_kw DECIMAL(10,2),
    smart_capabilities JSONB,
    status VARCHAR(20) DEFAULT 'ACTIVE'
);

CREATE TABLE charge_sessions (
    session_id UUID PRIMARY KEY,
    station_id UUID NOT NULL,
    port_id UUID NOT NULL,
    vehicle_id UUID NOT NULL,
    start_time TIMESTAMP NOT NULL,
    end_time TIMESTAMP,
    energy_delivered_kwh DECIMAL(10,2),
    cost_amount DECIMAL(10,2),
    status VARCHAR(20) DEFAULT 'ACTIVE'
);
```

🚗 Vehicle Service

Responsibilities:

- Vehicle registration and validation
- Owner management and profiles
- Vehicle type classification
- Historical parking pattern analysis

API Endpoints:

```http
POST   /api/v1/vehicles                    # Register vehicle
GET    /api/v1/vehicles/{id}               # Get vehicle details
PUT    /api/v1/vehicles/{id}/type          # Update vehicle type
GET    /api/v1/vehicles/{id}/history       # Parking history
GET    /api/v1/vehicles/search             # Search vehicles
```

💳 Payment Service

Responsibilities:

- Dynamic billing calculation
- Multiple payment method support
- Invoice generation and management
- Refund processing
- Pricing strategy management

API Endpoints:

```http
POST   /api/v1/payments/calculate          # Calculate parking fee
POST   /api/v1/payments/process            # Process payment
GET    /api/v1/payments/invoices/{id}      # Get invoice details
POST   /api/v1/payments/refunds            # Process refund
GET    /api/v1/payments/pricing-strategies # Get pricing options
```

3.3 Inter-Service Communication Patterns

Synchronous REST APIs:

python

```
class ParkingService:
    def validate_vehicle_registration(self, vehicle_id: str) -> bool:
        try:
            response = requests.get(
                f"http://vehicle-service:8080/api/v1/vehicles/{vehicle_id}",
                headers={'Authorization': f'Bearer {self.auth_token}'},
                timeout=3  # Circuit breaker timeout
            )
            return response.status_code == 200
        except requests.exceptions.Timeout:
            # Fallback logic or circuit breaker
            return False
```

Asynchronous Event-Driven Architecture:

python

```
@dataclass
class VehicleCheckedInEvent:
    event_id: str
    vehicle_id: str
    slot_id: str
    checkin_time: datetime
    lot_id: str
    vehicle_type: str

class ChargingEventHandler:
    def handle_vehicle_checked_in(self, event: VehicleCheckedInEvent):
        if event.vehicle_type == 'ELECTRIC':
            self.notify_charging_availability(event.lot_id, event.vehicle_id)
```

3.4 Database Per Service Strategy

Benefits:

- **Data Isolation**: Each service owns its data model
- **Independent Scaling**: Databases scale based on service load
- **Technology Freedom**: Each service can use optimal database technology
- **Failure Containment**: Database issues don't cascade across services

Implementation:

yaml

```
services:
  parking-db:
    image: postgres:14
    environment:
      POSTGRES_DB: parking_service
      POSTGRES_USER: parking_user

  charging-db:
    image: postgres:14
    environment:
      POSTGRES_DB: charging_service
      POSTGRES_USER: charging_user

  vehicle-db:
    image: mongodb:5.0  # Document store for flexible vehicle data
    environment:
      MONGO_INITDB_DATABASE: vehicle_service
```

3.5 API Gateway Configuration

yaml

```yaml
routes:
  - path: /api/v1/parking/**
    service: parking-service
    authentication: required

  - path: /api/v1/charging/**
    service: charging-service
    authentication: required

  - path: /api/v1/vehicles/**
    service: vehicle-service
    authentication: required

  - path: /api/v1/payments/**
    service: payment-service
    authentication: required
```

4. Architectural Evolution

4.1 From Monolith to Microservices-Ready

The refactoring journey transformed the system from a God Class anti-pattern to a clean, layered architecture ready for microservices decomposition:

Original Architecture Issues:

- God Class handling all responsibilities (GUI, business logic, data management)
- Broken inheritance hierarchies
- Tight coupling between components
- Poor error handling and user experience

Refactored Architecture Achievements:

- Clear separation of concerns (GUI, Business Logic, Data layers)
- Fixed inheritance with proper OOP principles
- Professional package structure and organization
- Comprehensive error handling and user feedback
- Scalable foundation for future extensions

4.2 Enhanced EV Charging Capabilities

The system was extended with sophisticated EV charging management including:

- Smart charging algorithms with load balancing
- Reservation systems and dynamic pricing
- Vehicle-to-Grid (V2G) support readiness
- Renewable energy integration capabilities
- Advanced analytics and business intelligence

4.3 Professional UI/UX Refinement

During final implementation, several user experience issues were identified and resolved to ensure the application meets professional software standards.

UI/UX Issues Addressed

Original Implementation Limitations:

- Maximize functionality disabled, restricting window management
- Application window positioned below taskbar, obscuring content
- No scrollbar in output console, limiting message history visibility
- Missing auto-scroll feature for real-time updates
- Inconsistent window positioning across systems

Technical Improvements Implemented:

Window Management Enhancements:

python

```
# Enhanced window configuration
root.geometry("800x950")  # Optimized dimensions
root.resizable(1, 1)      # Enabled maximize and resize functionality
```

Scrollbar Implementation:

python

```
# Added scrollbar to text console
scrollbar = tk.Scrollbar(text_frame)
scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
output_text = tk.Text(..., yscrollcommand=scrollbar.set)
scrollbar.config(command=output_text.yview)
```

Auto-scroll Feature:

python

```
# Ensure latest messages are always visible
def display_message(message):
    output_text.insert(tk.END, message + "\n")
    output_text.see(tk.END)  # Auto-scroll to bottom
```

Impact on User Experience

Aspect	Before	After	Improvement
Window Management	Restricted controls	Full window controls	Enhanced usability
Content Visibility	Messages lost off-screen	All content accessible	Data integrity
Output History	Limited view	Full scrollable history	Complete audit trail
Real-time Updates	Manual scrolling required	Auto-scroll to latest	Immediate feedback

Professional Standards Achieved

These refinements ensure the application adheres to standard desktop application conventions:

- Proper window management following platform guidelines
- Content accessibility through scrollable interfaces
- Real-time user feedback for all operations
- Consistent behavior across different display configurations

The UI/UX improvements demonstrate attention to both functional requirements and user-centered design principles, resulting in a professional-grade application suitable for production use.

5. Conclusion

This Domain-Driven Design analysis demonstrates a systematic approach to transforming a legacy parking management system into a scalable, enterprise-grade solution. The bounded context analysis, domain modeling, and microservices architecture proposal provide a clear roadmap for future evolution while maintaining the core functionality that users depend on.

The architectural decisions made ensure:

- **Scalability** through microservices decomposition
- **Maintainability** through clear domain boundaries
- **Extensibility** through well-defined APIs and interfaces
- **User Experience** through professional UI/UX refinements
- **Business Value** through advanced EV charging capabilities

This approach positions the Parking Management System for continued evolution while maintaining the robustness and reliability required for production deployment.