SWD-LB ITS PROJECT

Parking

Informationstechnik und System-Management / Wirtschaftsinformatik

Fachhochschule Salzburg GmbH

vorgelegt von

Miguel Alves

Pedro Jorge

Puch/Salzburg, September 2025

Inhaltsverzeichnis

1 Project Description

1.1 Project Setting

The Portuguese Government recently acquired a 2-story abandoned parking space in the center of Lisbon and is planning on remodeling it. After being awarded the public contract the company LisboaBoys is working on a project that will make operational the recently acquired space.

1.1.1 Current situation

The space was left clean by the Portuguese Government so that the company LisboaBoys can scout and design the project according to the scope, specifications and other similar products.

Internally LisboaBoys was a medium sized team of 2 programmers, 1 designer and one project manager that were assigned to work in this project.

In accordance with the agreements made the Portuguese Government would be responsible for providing:

- An API for a payment application,
- Algorithm for license plate extraction using camera,
- Algorithm for recognizing disability placards/markers when visibly displayed on the vehicle's windshield,
- Scheduling API to guarantee the parking spaces are not assigned to multiple customers at the same time,
- The necessary permits for operations,
- Guarantying security to the space and the property inside-it,
- Electricity and water usage,
- An active and maintained server for the use of the application,
- Materials/hardware according to necessities if provided with written explanations for necessity.

1.1.2 Project goals

Goals for this project consist of:

- Deploy a new parking solution to improve day-to-day operations and maintenance,
    - Modernize parking operations with automated gates, license-plate recognition, and centralized control,
    - Streamline both entry and exit processes for customers,
    - Reduce manual workload and operational errors.
- Provide an extensible, server-centered solution that supports any number of parking gates and payment devices,
    - Central server handles all business logic, data management, and communication with gates and payment devices,
    - System supports any number of gates and devices without requiring redesign,
    - Extensible architecture allows future integration with new, integration of more parking spaces and integrating other parking buildings.
- Manage state of the parking area in real time,
    - Track availability of parking slots at any moment,
    - Synchronize state changes across multiple gates concurrently,
    - Prevent inconsistencies such as double-booking.
- Manage customer information/ applications and information necessary to produce statistics,
- Manage the different types of tickets and different types of parking spaces,
- Collect and analyze parking data to produce statistics and support for future planning / marketing and escalation purposes,
- Manage master data, such as:
    - Customer and vehicle information
    - Number of parking spaces in general, per type and their IDs,
    - Number and occupation statistics for parking spaces,
    - Yearly statistics and analysis.
- Manage transactional data, such as:
    - Ticket validity and type,
    - Ticket assignment,
    - Contracts,
    - Pricing,

- Movements in and out of the parking spaces,
- Real Time occupancy.

1.1.3 No Project goals

We decided to divide the non-project goals into 3 categories, namely Hardware and infrastructure, Business and marketing, Performance and enterprise-level. We chose these categories because we considered that those were the groups in which the topics that we identified and that could be a problem we weren't able to solve would fall in.

In the group Hardware and infrastructure, we relate topics that are outside the project competencies, at the current state, noting that adding any of the topics to the project scope would throw of the balance of the topic and require further investment of time and money. An example being the development of our algorithm to extract the license plate out of the image produced with the camera. While others would take a step forward to being impossible using as an example the creation of our payment platform. Some of the topics may also fall under services we cannot provide for being outside our business model, for example providing on-site security and providing cyber security.

In the group business and marketing and Performance and enterprise-level ops we find other things that not only fall outside our "know how" but are also matters that should be decided by the owner of this project, the Portuguese Government.

Hardware and infrastructure:

- Integration with real physical gates, sensors, or payment,
- Camera hardware management,
- Networking and physical server deployment,
- Development or training in license plate recognition or accessibility placard detection algorithms,
- Implementation of a real payment platform or handling sensitive financial transactions,
- Compliance with banking or other regulations,
- Customer behavior and enforcement,
- Enforcement of misuse, for example occupying the wrong parking space and staying after the end of the defined time,
- On-site monitoring, security guards, fines or any staff,
- Legal and administrative aspects,
- Management of permits, parking regulations, or governmental approvals.

Business and marketing:

- Developing full marketing strategies to attract occasional customers or convert them into regular contracts,
- Customer service, dispute resolution, or billing complaints handling.

Performance and enterprise level:

- Scaling to enterprise-level loads,
- Disaster recovery and backup strategies,
- Develop a mobile compatible application.

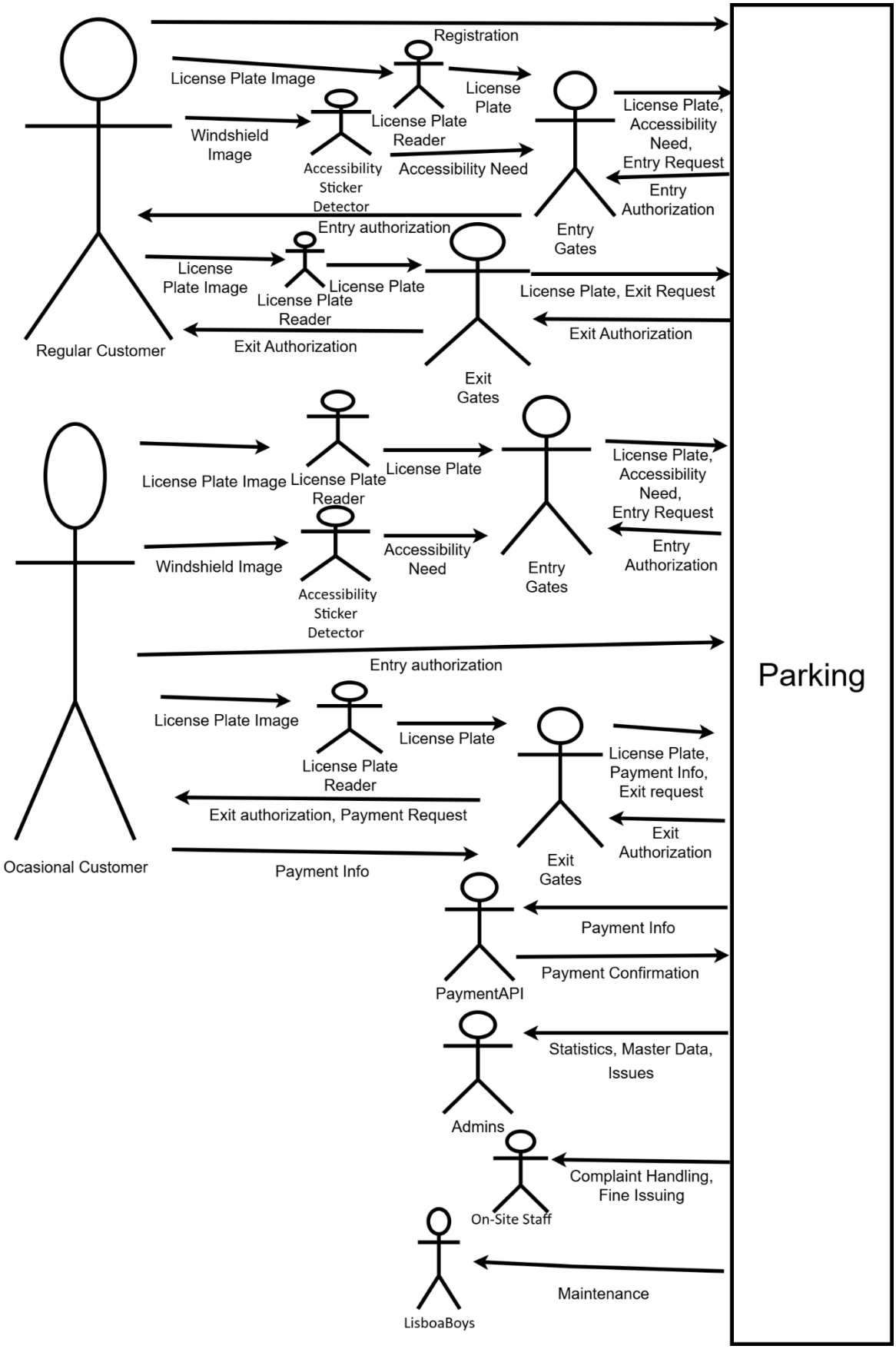
1.1.4 List of Stakeholder

For the stakeholders of this project and considering that we are working with a government we decided to include everyone that will interact with this project as we value information and experience coming from either the top with the Portugues government or the bottom with the customers.

- Portuguese Government:
    - Owner of the parking infrastructure and primary sponsor. Responsible for financing, overall governance, and compliance with national regulations.
- LisboaBoys:
    - Company is responsible for day-to-day operation and maintenance of the parking facility. Ensures that the system supports business processes and provides reliable service.
- Regular Customers:
    - Customers with long-term contracts. Require reliable access to their reserved slots and seamless entry/exit without on-site payments.
- Occasional Customers:
    - Ad-hoc users of the parking area. Their experience depends on smooth entry and exit, accurate billing, and availability of suitable slots.
- System Administrators:
    - Internal staff are responsible for maintaining master data, monitoring usage statistics, and troubleshooting technical issues.
- Parking Attendants / On-Site Staff:
    - Even if the system is largely automated, staff may need to intervene in exceptional cases, handle customer complaints, or override the system in case of failure.
- Payment Service Provider:
    - External service for processing payments. In the project, this will be simulated, but in a real-world scenario it represents critical financial infrastructure.
- Accessibility and Disability Advocacy Groups:
    - Stakeholders ensuring that accessible parking slots are correctly reserved, priced fairly, and available to eligible users.

- Regulatory and Municipal Authorities:
    - Authorities are interested in compliance with laws and safety standards.
- Competitors:
    - Even if they are negatively affected, we must also consider them in the design of our project to “one up” them and reach a larger number of costumers.

1.2 Context Diagram



1.3 Requirements

1.3.1 Functional requirements

The project's requirements document already describes the functional requirements. Here are a recap and a clarification of the requirements according to our understanding:

Gate Operations

- Each gate must communicate with the central server to:
    - Query available/free parking spaces,
    - Notify vehicle entry including license plate and accessibility marker detection,
    - Request billing information before vehicle exit,
    - Notify vehicle exit and release the slot.

Parking Slot Management

- The system distinguishes slot types: simple, extended including accessible, and oversize,
- Accessible slots must be reserved exclusively for eligible drivers,
- Occupancy state of each slot must be maintained in real-time,

Customer Management

- Support for regular customers with contracts and reserved slots,
- Support for occasional customers with ad hoc usage and per-use billing,
- Registration of customer data and associated vehicles using the license plates as an identifier.

Ticket and Contract Handling

- Two ticket types:
    - Season ticket,
    - Single-use tickets.
- Tickets must be validated at gate,
- Grace periods after payment or expiry must be enforced.

Payments and Billing

- Occasional customers: per-use billing according to slot type and eligibility,
- Regular customers: prepaid contracts with electronic billing,
- Every payment must be linked to exactly one ticket, with timestamp and amount recorded,
- Concurrency control must avoid race conditions during payments and exits.

Statistics and Reporting

- Provide usage statistics by day, month, and year,
- Track movements of each vehicle with entries and exits with timestamps,
- Maintain monthly usage statistics for each season ticket,
- Support marketing analysis.

1.3.2 Non-functional requirements

1.3.2.1 Quality requirements

- Availability
    - The system must operate continuously, 24 hours a day, 7 days a week, since the parking facility is designed to function automatically without requiring permanent on-site staff.
- Usability
    - Interfaces must be intuitive and require minimal user training,
    - The system must be designed to impede double booking.
- Reliability
    - The system should handle unexpected interruptions gracefully by providing clear error messages or fallback mechanisms.
- Security
    - While deep cybersecurity measures are outside the scope, the system must ensure that only authorized operations are processed.

1.3.2.2 Constraints

- External dependencies:
    - The project depends on external services provided by the Portuguese Government, including the payment API, license plate recognition, disability placard recognition, server, database and cybersecurity specialists.
- Team capacity:
    - Development is constrained by the limited size of the LisboaBoys project team. This limits the number of use cases and the complexity of the implementation.
- Time constraint:
    - The project must be fully implemented and demonstrated to the Portuguese Government by the agreed deadline.
- Budget constraint:
    - The project operates under a limited budget defined by the Government. No additional funds are available for developing proprietary hardware, payment platforms, or large-scale infrastructure.

2 Behavior

2.1 Overview Use Case Diagram

2.2 Use Case 1 Purchase a season ticket

2.2.1 Use Case Description

Use Case ID:	1		
Use Case Name:	Purchase a season ticket		
Created By:	Miguel Alves	Last Updated By:	Miguel Alves
Date Created:	01/10/2025	Date Last Updated:	15/10/2025
Actors:	Customer		
Description:	The customer purchases a season ticket by selecting a parking slot from available slots. The system verifies availability, reserves the slot, creates and assigns the season ticket and presents the price.		
Trigger:	Customers want to purchase a season ticket.		
Pre-Conditions:	1. To be able to purchase a season ticket the user must first have to log in. 2. To log in the customer must first register himself. 3. Available parking spaces must be present in the system.		
Post-Conditions:	1. A new season ticket is created and stored. 2. The ticket is linked to the customer, license plate and reserved slot. 3. The reserved slot is blocked for the duration of the ticket validity.		
Normal Flow:	User registers into the system. User logs into the system. System verifies login credentials. User selects Purchase Season Ticket. System displays available parking slots from cache.		

	User selects a slot. System verifies slot availability. System presents price for the selected slot. User confirms purchase. System creates season tickets and assigns it to the user, license plate and slot. System finalizes and shows confirmation.
Alternative Flows:	No account: • User logs into the system. • User not found. • System suggests the user to register. Not logged in: • User selects Purchase Season Ticket. • The session has expired. • The system asks the user to log in and try again. Concurrent selection of slots: • User registers into the system. • User logs into the system. • System verifies login credentials. • User selects Purchase Season Ticket. • System displays available parking slots from cache. • User selects a slot. • (Concurrent user finalizes creation of the ticket using this slot) • (Cache is updated) • System verifies slot availability. • System shows error messages and tells the user try again. • System displays available parking slots from cache. • User selects a slot. • System verifies slot availability. • System presents price for the selected slot. • User confirms purchase. • System creates season ticket and assigns it to the user, license plate and slot. • System finalizes and shows confirmation.

	Failure to finish payment: • User registers into the system. • User logs into the system. • System verifies login credentials. • User selects Purchase Season Ticket. • System displays available parking slots from cache. • User selects a slot. • System verifies slot availability. • System presents price for the selected slot. • User confirms purchase. • Payment fails. • Message indicates the user to try again or cancel order. • User tries again. • Payment goes through. • System creates season ticket and assigns it to the user, license plate and slot. • System finalizes and shows confirmation. Order cancellation: • User registers into the system. • User logs into the system. • System verifies login credentials. • User selects Purchase Season Ticket. • System displays available parking slots from cache. • User selects a slot. • System verifies slot availability. • System presents price for the selected slot. • User confirms purchase. • Payment fails. • Message indicates the user to try again or cancel order. • User cancels order. • System receives the order and says goodbye to user. • System marks previously selected slot as available.
Exceptions:	Payment failure / timeout: • If the payment fails or times out, cancel the purchase, release the slot and log the event. Transaction/locking failure: • If the creation of the ticket and the slot reservation cannot be committed atomically (ex: unique constraint violation or lock

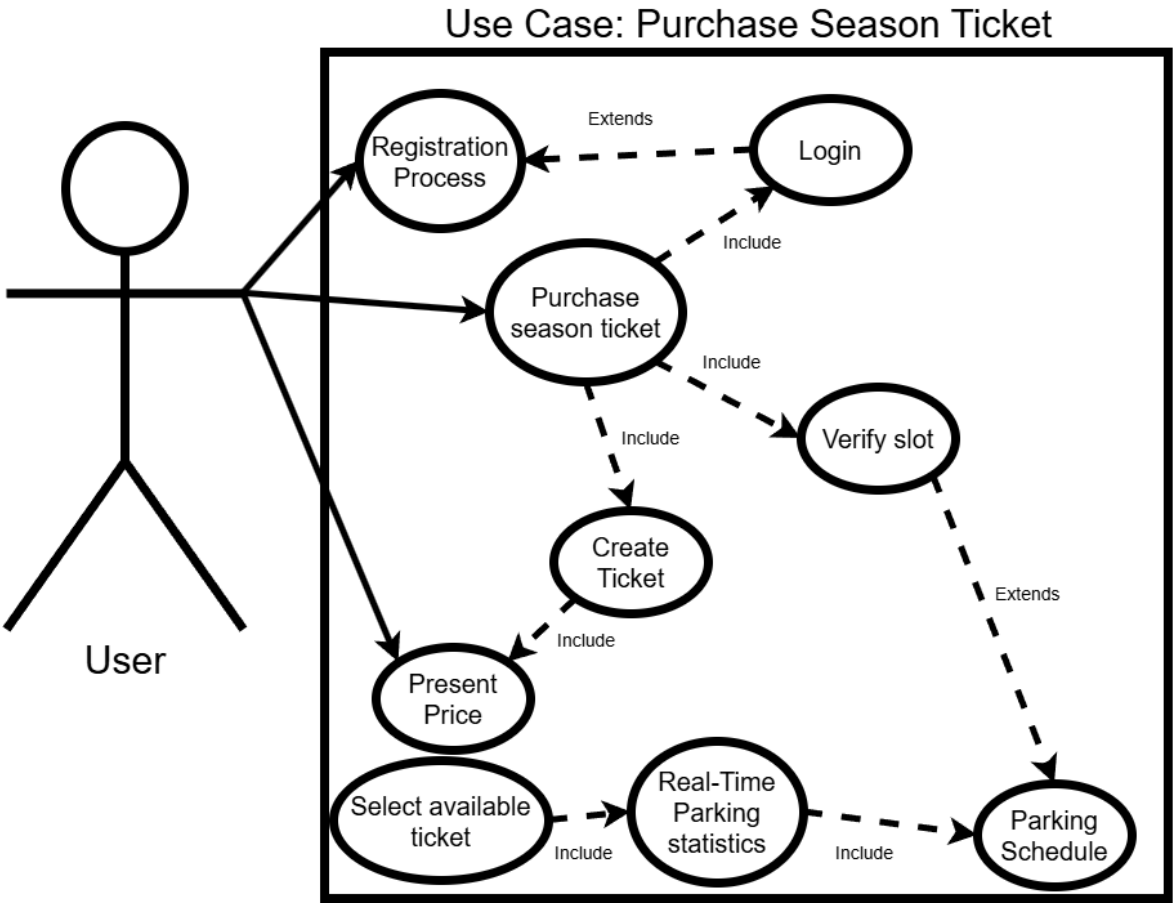
	timeout), the system performs a rollback, informs the user and displays the updated slot list. External services unavailable: • If required external services (ex: pricing service, identity service) are unavailable, the system fails gracefully: it shows the message "Service temporarily unavailable," does not create the ticket and does not block the slot. System crash: • In case of an unexpected failure, any partially created ticket is rolled back, the slot is released. The event is logged and an alert is raised for analysis.
Includes:	Register User: • Required if the customer does not yet have an account. Login User: • Required before any purchase action can be performed. Verify Slot Availability: • Common functionality to check if the chosen parking slot is still available before finalizing the purchase. Process Payment: • Common functionality used in any use case requires a payment step. Assign License Plate: • Ensure that the purchased ticket is linked to the correct vehicle and plate. Generate Confirmation / Receipt: • Standard step to finalize the transaction and provide a record to the customer.
Priority:	Hight • Purchase requires a registered customer • System creates season tickets and assigns customers + license plate + slot • Reserve the selected slot for the whole validity period • Multi-user correctness (no double booking; concurrent access) Medium • Customer selects a slot of desired type from currently available • System verifies chosen slot is still available (at commit time)

	Low • System presents applicable price at an appropriate step
Frequency of Use:	Expected to be medium. Compared to single/occasional tickets, season tickets are purchased less frequently (ex: once per customer per season or per contract period). However, for a large parking facility, this use case will be performed several times per week, particularly at the beginning of each billing cycle or season.
Business Rules:	A customer must be registered and logged in to purchase a season ticket. Each season ticket must be linked to one license plate and one parking slot. A slot reserved by a season ticket is blocked for the entire validity period of the ticket. Prices for season tickets are determined by the pricing policy defined in the system (ex: based on slot type, accessibility, or vehicle type). Only available slots can be purchased; the system must prevent double-booking. All payments must be confirmed through the payment processing service (mock API) before the ticket is created and finalized.
Special Requirements:	Concurrency, Consistency and Atomicity: • The creation of the season ticket, slot reservation and license plate assignment must be atomic; either all succeed or none. • The system must prevent double booking of the same slot. • If a conflict occurs (ex: slot already taken), the operation fails gracefully and presents the user with an updated slot list. Reliability and Fault Tolerance: • If the payment service fails or is unavailable, the purchase is canceled and the slot is released. • If external services (ex: pricing, identity) are unavailable, the system must fail safely, show a clear error message and not block the slot. • In case of a system crash, partial transactions are rolled back and slots are returned to the free state.

	Security and Privacy: • Only authenticated customers can access the purchase flow; each critical step requires a valid session. • Each ticket must be bound to exactly one customer and one license plate. Usability: • Error messages must be clear and actionable (ex: informing the user that a slot is no longer available and offering a refresh). Data Validation and Quality: • The system must validate slot type compatibility, correct license plate format and prevent multiple active season tickets for the same plate.
Assumptions:	1. Third-party and external systems, such as the mock payment API, pricing service and authentication service are available and functioning correctly during normal operation. 2. The parking hardware is correctly installed and provides valid data to the system. Hardware malfunctions are outside the project's scope. 3. Customer data such as registration and license plate information is accurate and up to date when entered into the system. 4. Network connectivity between system components server, client application and the database are stable and reliable. 5. The database is operational and configured according to project specifications, with integrity constraints already defined. 6. Concurrent users will be limited to a reasonable number consistent with expected real-world usage. Stress or load testing beyond this is outside scope. 7. Authentication credentials are managed by the identity service, not by this use case implementation. 8. The LisboaBoys system and the Portuguese Government infrastructure are assumed to cooperate under stable contractual and operational conditions.

	9. The system is deployed in an environment with adequate security, electricity and network infrastructure, as provided by the government.
Notes and Issues:	

2.2.2 Activity Diagram



2.3 Use Case 2 Entry with Season Ticket

2.3.1 Use Case Description

Use Case ID:	2		
Use Case Name:	Entry with Season Ticket		
Created By:	Pedro Jorge	Last Updated By:	Pedro Jorge
Date Created:	02/10/2025	Date Last Updated:	13/10/2025
Actors:	Regular Costumer		

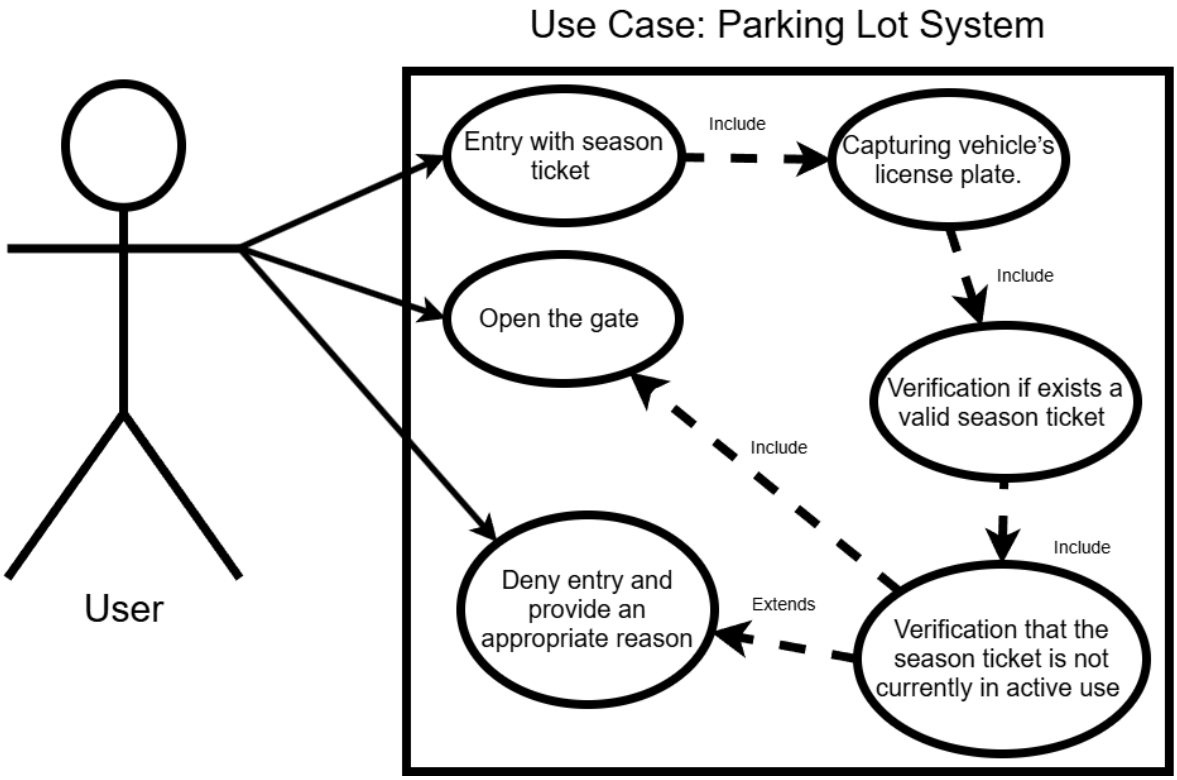
Description:	The camera at the entry gate captures the vehicle's license plate. The system verifies if there is a valid season ticket associated with that license plate and then confirms that the ticket is not currently in active use. If both checks are successful, the gate opens and the entry is recorded. Otherwise, entry is denied, and an appropriate reason is provided.
Trigger:	The customer drives their vehicle to the parking lot's entry gate.
Pre-Conditions:	• The customer must have a valid season ticket associated with the vehicle's license plate. • The season ticket must not be in use (i.e., the vehicle cannot enter without a corresponding exit). • The camera system and server must be functioning properly.
Post-Conditions:	• The system records a new movement with the entry timestamp and gate identifier. • The ticket's status is marked as "in use". • The monthly usage count for the season ticket is updated. • If the verification fails, the gate remains closed, and entry is denied.
Normal Flow:	2.1. The customer drives to the entry gate. 2.2. The entry gate camera captures the vehicle's license plate. 2.3. The system sends the vehicle's license plate to the central server. 2.4. The server verifies if there is a valid season ticket associated with that license plate. 2.5. The server checks that the ticket is not already "in use". 2.6. The system records the vehicle's entry, including the timestamp and gate. 2.7. The system instructs the gate to open. 2.8. The customer enters the parking lot.

Alternative Flows:	Invalid Ticket: 2.1.1. The customer drives to the gate. 2.1.2. The camera captures the license plate. 2.1.3. The system checks the license plate but does not find valid season ticket. 2.1.4. The system denies entry and displays a message at the gate (e.g., "Invalid ticket”). Ticket Already in Use: 2.2.1. The customer drives to the gate. 2.2.2. The camera captures the license plate. 2.2.3. The system checks the license plate and finds a valid season ticket. 2.2.4. The system verifies the ticket's status and detects that it is already "in use”. 2.2.5. The system denies entry and displays a message at the gate (e.g., "Ticket already in use").
Exceptions:	License Plate Read Failure: • The camera fails to read the vehicle's license plate (e.g., due to being dirty, damaged, or poor lighting). • The system notifies the customer that the read failed and, if applicable, provides an alternative (such as a button to contact support). The gate does not open. Verification Service Unavailable: • The gate is unable to communicate with the central server to verify the season ticket. • The system displays an error message ("Service temporarily unavailable") and the gate remains closed. The event is logged for analysis.
Includes:	Capture License Plate: The functionality to capture the license plate is a necessary step to initiate the entry process. Verify Valid Season Ticket: Verifying the ticket's validity is a crucial functionality that must be executed to permit entry. Verify Ticket Not in Use: Checking the ticket's status (if it's already in use) is a required functionality to ensure system consistency. Record Movement: Recording the vehicle's entry is an essential step for operational tracking and reporting

Priority:	• The camera captures the vehicle's license plate: Must, it depends on R1 • System verifies for a valid season ticket associated with the license plate: Must, it depends on R2 • System verifies that the ticket is not currently in active use: Must, it depends on R2 • If checks pass, system opens the gate and records a movement.: Must, it depends on R3, R4 • If checks fail, system denies entry and provides an appropriate reason: Must, it depends on R3, R4
Frequency of Use:	High: This use case is expected to be performed very frequently, potentially multiple times a day by a single user. For the entire parking facility, it will occur dozens or maybe even hundreds of times daily, making it a highly trafficked use case. This is one of the most common user interactions with the system, along with the exit flow.
Business Rules:	• The system requires a valid ticket to pass any gate. • A season ticket shall allow unlimited entries and exits within its paid validity period. • Each season ticket is bound to exactly one parking slot and a specific license plate. • The system must deny entry if no valid season ticket is associated with the license plate or if the ticket is already in active use. • The system shall record individual movements. A movement shall consist of the vehicle's license-plate identifier, the ticket, and an entry timestamp and gate identifier.
Special Requirements:	Performance: The entire process from license plate capture to gate opening must be extremely fast to prevent vehicle congestion at the gate. The system needs to handle concurrent entry attempts from multiple gates simultaneously without performance degradation. Concurrency and Consistency: The system must ensure that only one vehicle can be using a specific season ticket at any given time. This requires an atomic check-and-update operation on the ticket's "in use" status to avoid race conditions. Reliability: The system must be robust enough to handle communication failures between the gate hardware and the central server. In case of a temporary connection loss, the gate should have a defined fail-safe behavior (e.x., deny entry, log the event, or use a local cache if available). Fault Tolerance: If the central server is unavailable, the system should

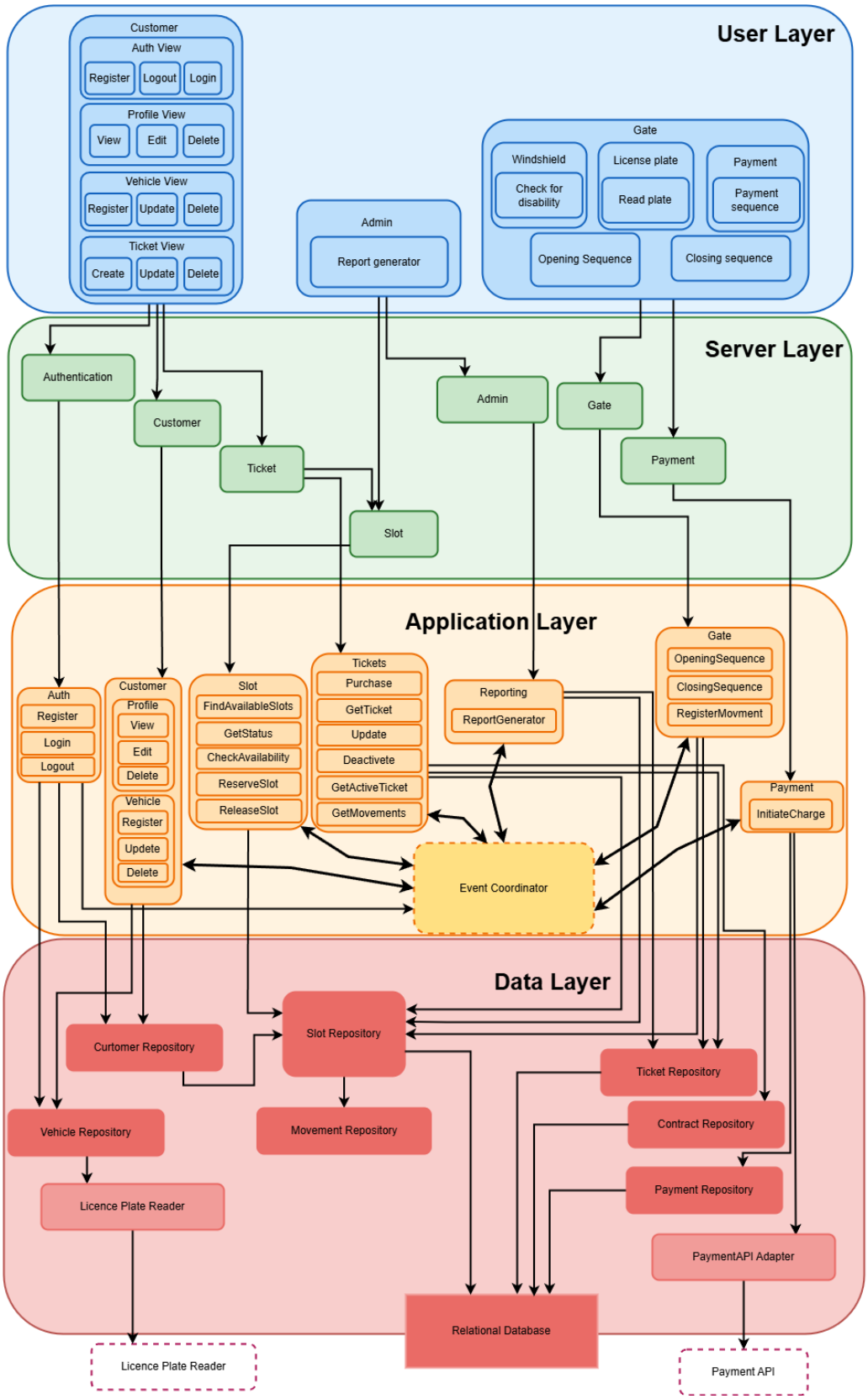
	show a "Service temporarily unavailable" message at the gate and not allow entry to avoid data inconsistencies. Data Validation: The system must be able to handle cases where the license plate read is incomplete or unreadable, triggering an error state without crashing.
Assumptions:	• The license plate capture mechanism at the gate is reliable and provides accurate data to the system. • The central server maintains a real-time occupancy state of the parking area. • The system correctly manages the "in use" state of a season ticket. • Network connectivity between the gates and the central server is stable and reliable. • The system's central database is operational and its integrity is maintained.
Notes and Issues:	Issue: How does the system handle an unregistered or invalid license plate being driven to the gate? The use case description only covers customers with season tickets. Resolution: The system should log the failed attempt and, for potential future use, display a message that guides the driver (e.g., "Take a ticket" for occasional customers). Issue: What happens if a vehicle enters but the system fails to record the movement? Resolution: The system must have a robust logging and error-handling mechanism. Any failure in recording a movement should be flagged as an alert for administrative review. Issue: Can the same season ticket be linked to multiple license plates? Resolution: Based on the requirement that a ticket is bound to "the specified license plate", we assume a 1:1 relationship between a season ticket and a single license plate. If multiple plates are needed, this should be an additional use case or a modification to the system.

2.3.2 Activity Diagram



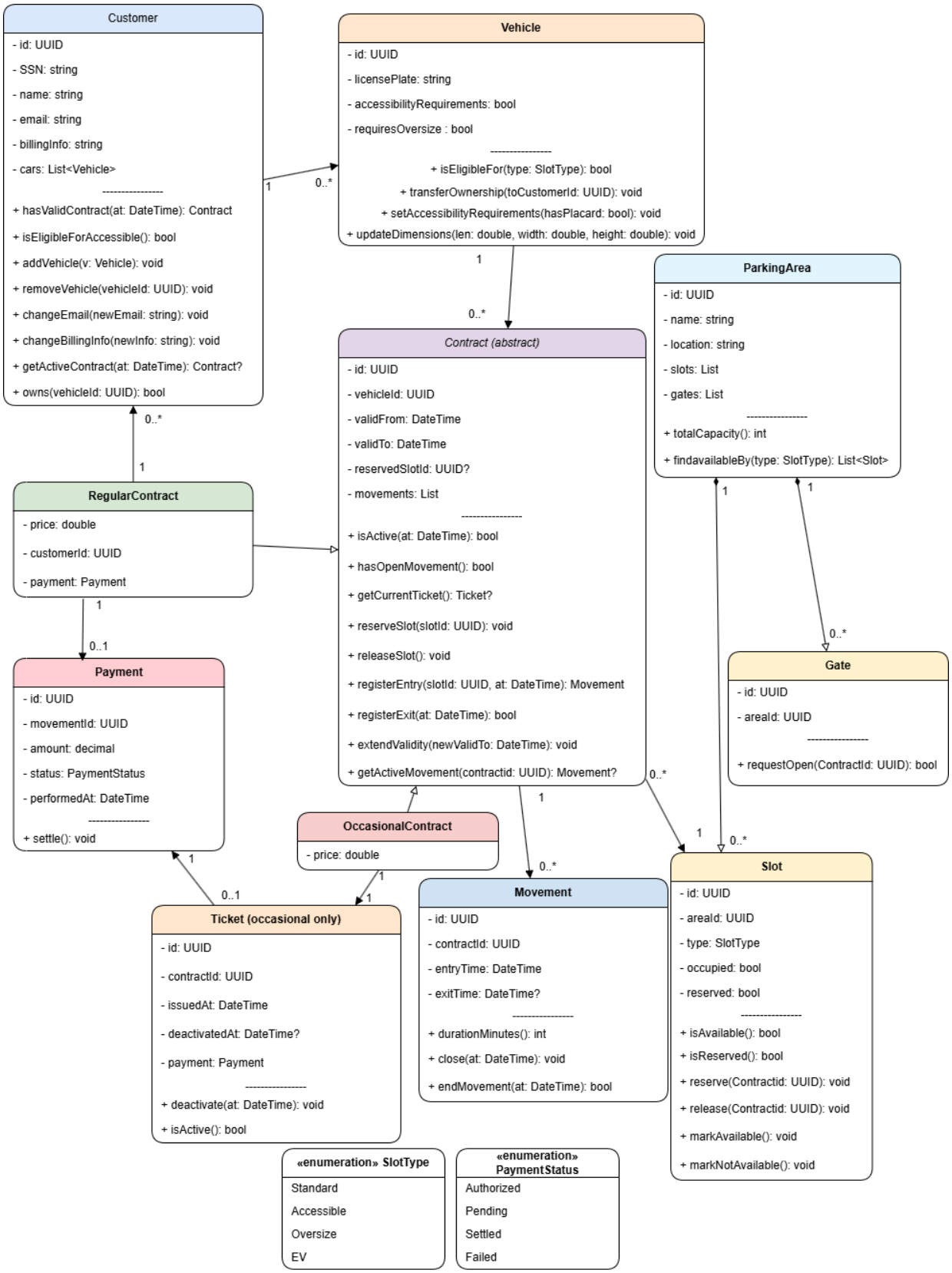
3 Architecture

3.1 Component Diagram

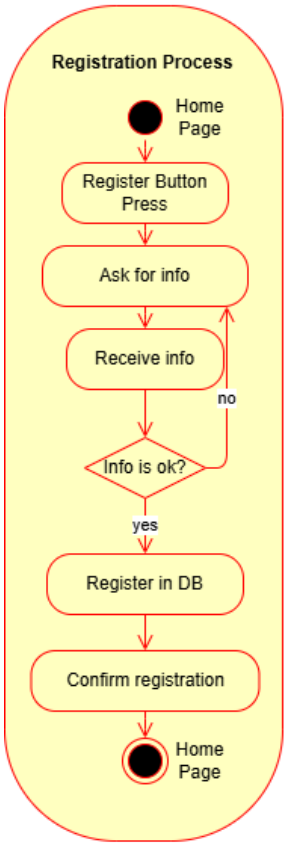


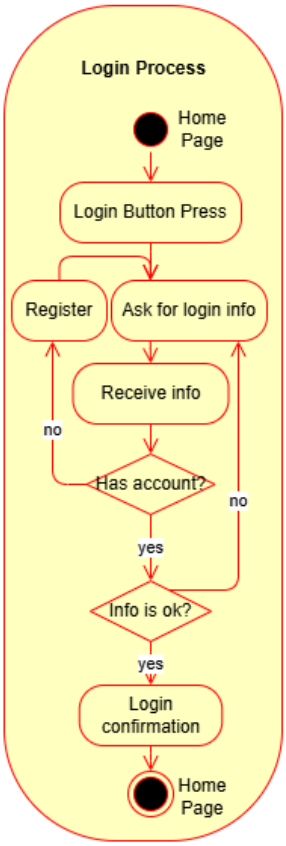
4 Implementation

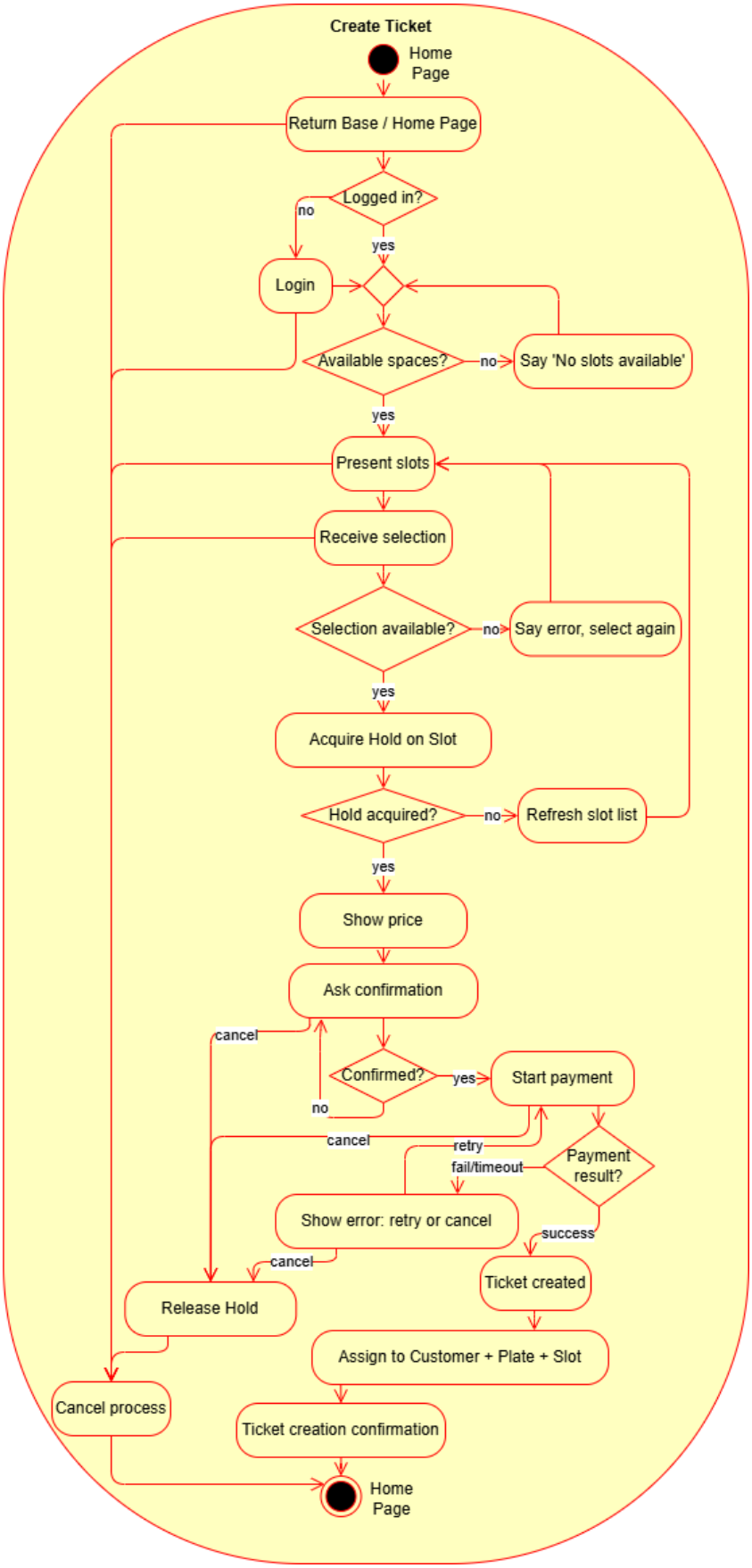
4.1 Class Diagrams



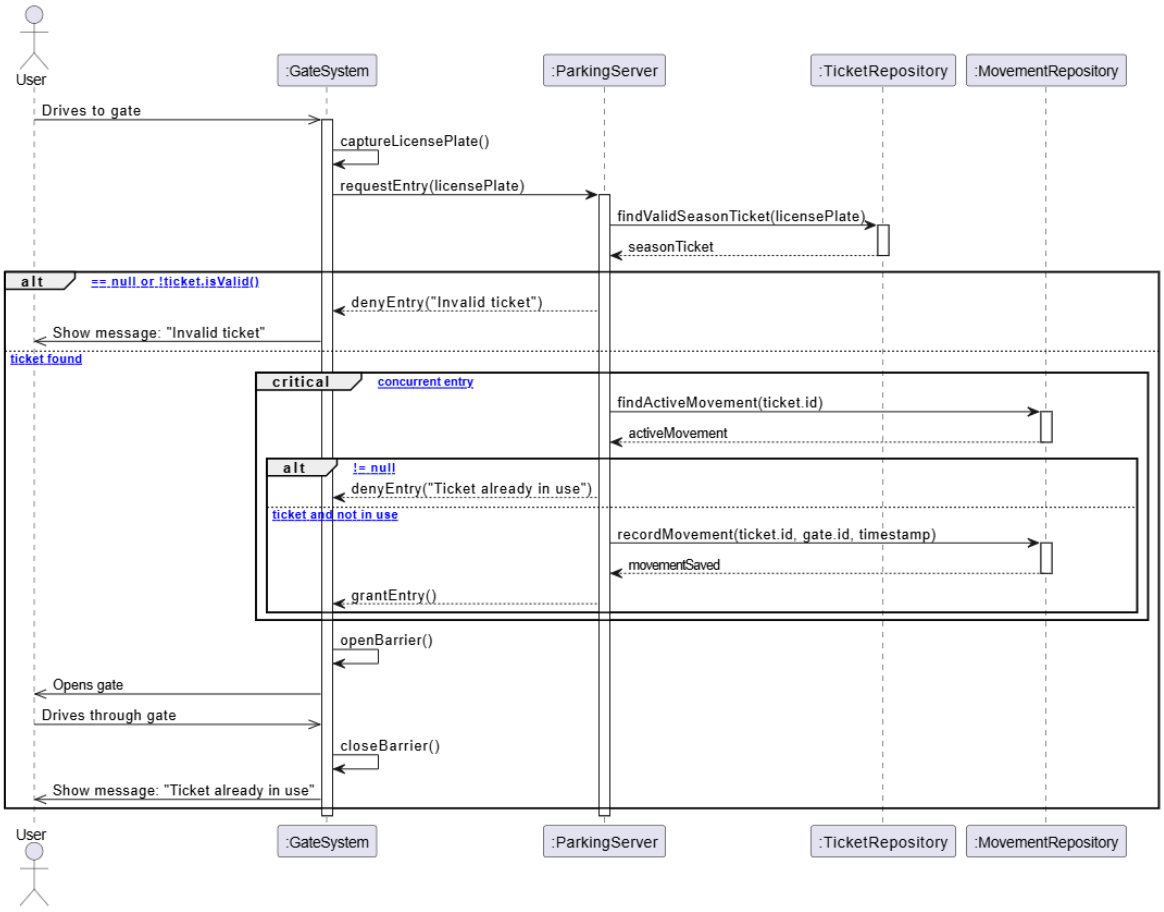
4.2 Activity Diagrams







4.3 Sequence Diagrams



5 Implementation

This section describes the concrete implementation of the PortugueseParking system, focusing on the project structure, domain model, business logic, and testing approach. While earlier chapters focused on conceptual diagrams and requirements, this section explains how those concepts were translated into a working Django application with real business logic, data handling, and user interface behavior. The goal of this implementation phase was to deliver a functional prototype covering the required use-cases UC1–UC3.

5.1 Project Structure

The project follows a classic Django multi-app architecture. Each domain area is encapsulated in its own Django app to enforce modularity and separation of concerns:

PortugueseParking/

- contracts/
  – Business rules for season tickets, occasional tickets, and movements
- customers/
  – User accounts and authentication
- parking/
  – Parking areas, slot types, slots, gates
- vehicles/
  – Vehicle entity and vehicle-related utilities
- swd_django_demo/
  – Project configuration, settings, dependency wiring

Each app contains its own:

- models.py with domain entities and persistence logic
- views.py with UI controllers for each use-case
- services.py with business logic (ticket and pricing services)
- tests/ with isolated unit tests and UI-level tests

This structure mirrors the logical decomposition of the system into four bounded contexts: **User, Parking, Vehicle, and Contracts**. It also aligns with the layered architecture proposed in the design phase, separating the domain model, application services, and presentation logic.

5.2 Domain Models Overview

The Django implementation closely follows the class diagrams included in the report, extending them with practical fields and behaviors required for a fully functioning system. The domain model is split across several apps:

- **customers**: user identities and authentication
- **parking**: physical structure of the parking facility
- **vehicles**: vehicle-related attributes and rules
- **contracts**: tickets, contracts, movements, and payments

The following subsections describe the most important parts of each domain area.

5.3 Customers Module

The customers app defines the user base of the system. It uses Django's AbstractUser as the base class to support full authentication while adding additional domain attributes:

- SSN (optional identification field)
- Billing information
- Credit balance (optional; not directly used in UC1–UC3 but kept for extensibility)

This module is responsible for:

- User registration and authentication
- Session management (login / logout)
- Associating vehicles and contracts with specific users

All use-cases that involve UI interaction (UC1 and UC2, and the operator-facing parts of UC3) require the user to be authenticated. This ensures that contracts and vehicles are always linked to a concrete customer account.

5.4 Parking Module

The parking app contains the static structural description of the physical parking facility. It includes the following key entities:

ParkingArea
Represents a physical area or floor inside the car park (e.g. *"Level -1"*, *"Outdoor Zone A"*). It is master data and changes infrequently.

SlotType
Defines the characteristics of a slot (e.g., Simple, Extended, Oversize, Accessible). It includes a size_rank which establishes a hierarchical compatibility rule for vehicles. This allows the

system to express that an oversize slot can host any smaller vehicle, while a simple slot cannot host an oversize vehicle.

ParkingSlot
Models the actual parking position in an area. It includes:

- A reference to a ParkingArea
- A SlotType
- An accessibility flag (for disabled users)
- Logic for verifying:
    - whether a slot is compatible with a vehicle (is_compatible_with)
    - whether a slot is free for a given time period (is_free_for_period)

This logic ensures that there is no double-booking of the same slot and that accessibility and size rules are respected.

Gate
Represents an entry/exit gate associated with a parking area. Gate instances are used in UC2 and UC3 to identify where a movement or occasional ticket entry/exit is triggered. Although Gate does not contain complex logic itself, it is a key part of the interaction between UI and service layer.

Overall, the parking module provides the spatial and structural backbone for the contract and ticket logic.

5.5 Vehicles Module

The vehicles app models the user's vehicles. A vehicle includes:

- License plate
- Owner (Customer)
- Minimum required SlotType
- Disability permit indicator

This module enforces compatibility rules between vehicles and parking slots and provides helper methods such as:

- requires_slot_type – returns the minimum SlotType required by a vehicle
- can_use_slot – delegates compatibility checks to the ParkingSlot logic
- active_contracts – filters the contracts associated with a vehicle for a given time interval

Vehicles form the bridge between the customer domain and the contract domain, allowing season tickets to be tied to specific license plates, while also enforcing the size and accessibility constraints described in the requirements.

5.6 Contracts Module

The contracts app is the central functional component of the project. It implements all the logic related to:

- Season tickets (UC1 + UC2)
- Occasional tickets (UC3)
- Parking movements
- Payments

The module contains several key domain entities that are largely derived from the earlier class diagrams, but extended with additional fields to support real-world behavior.

5.6.1 Contract Model Hierarchy

The base class Contract holds the attributes shared by all contract types:

- Vehicle reference
- Validity interval (valid_from, valid_to)
- Reserved slot
- Aggregated movements (one-to-many relationship)

Two concrete subclasses extend this base class:

RegularContract
Used for season tickets and directly corresponds to the “season ticket” concept in UC1 and UC2. It adds:

- Price
- Customer reference (link to the authenticated user)
- Payment reference (optional, pointing to a Payment entity)

This model is responsible for reserving a parking slot for a given period and validating whether it is currently active.

OccasionalContract
A simplified single-use contract model, used internally for occasional tickets. It is less central than RegularContract but provides a consistent place for tracking occasional parking agreements if needed.

5.6.2 Movement

Movement represents an individual parking session associated with a contract. It contains:

- Entry time
- Exit time
- Methods to compute duration in minutes
- A simple is_open helper (exit_time is null)

Movements are essential for UC2, as they model the “ticket in use” concept: a contract with an open movement corresponds to a vehicle that is currently parked in the facility.

5.6.3 OccasionalTicket

The OccasionalTicket model implements the logic of single-use, anonymous parking and is a key part of UC3. It includes:

- License plate
- Assigned slot
- Entry and exit timestamps
- Amount due and amount paid
- Payment timestamp
- Exit deadline (grace period after payment)
- Closure flag (is_closed)

Based on the functional requirement for “single-use tickets”, the implementation extends the original specification with full billing state management and enforcement of a configurable grace period. Tickets are fully anonymous (no Customer reference), but they are bound to a specific license plate and slot to support realistic behavior of a cash device in a parking garage.

5.6.4 Payment

The Payment model tracks the lifecycle of a payment associated with a contract or ticket. It includes:

- Amount
- Status (authorized, pending, settled, failed)
- Timestamp for when the payment was performed

RegularContract may reference a Payment entity directly, allowing the system to track whether UC1 completed successfully. For occasional tickets, payment information is stored

directly on the OccasionalTicket instance (amount_due, amount_paid, paid_at), as these tickets are anonymous and simpler.

5.7 Business Logic Layer (TicketService)

The TicketService class is the central piece of the business logic and corresponds to the Application Layer. It is responsible for orchestrating the use cases and interacting with:

- The pricing service (PricingService)
- The payment service (PaymentService)
- The persistence layer (repositories based on Django ORM managers)

All collaborators are injected via the ITicketService interface, which allows the service to be tested in isolation without a real database. By default, the service uses the actual ORM managers (e.g., ParkingSlot.objects), but unit tests can provide mocked repositories to validate behavior.

The TicketService encapsulates all three main use cases.

5.7.1 UC1 – Purchase Season Ticket

The method purchase_season_ticket(...) implements the full UC1 workflow:

- Load the vehicle and requested slot by their identifiers, checking that the vehicle belongs to the authenticated customer.
- Verify that the slot is not already reserved during the requested period by detecting overlapping contracts.
- Compute the total price using PricingService.get_season_price(...).
- Call PaymentService.process_payment(...) to charge the customer.
- If payment fails, the operation is aborted and no contract is created.
- If payment succeeds, a RegularContract is created, reserving the slot for the given period and linking it to the customer and vehicle.

All of this is wrapped inside a database transaction (transaction.atomic) and uses select_for_update to lock the relevant rows, preventing race conditions and ensuring that two customers cannot reserve the same slot concurrently for overlapping periods.

5.7.2 UC2 – Entry/Exit with Season Ticket

The service implements both entry and exit flows for season tickets.

Entry – enter_with_season_ticket(license_plate, gate_id)

- Normalize and look up the vehicle by its license plate.
- Find an active RegularContract for the current moment in time.

- Verify that there is no open Movement for that contract (season ticket not already “in use”).
- Validate that the gate exists.
- Create a new Movement with the current timestamp as entry_time.
- Return a structured response including flags such as success and open_gate, as well as explanatory messages in reason.

Exit – exit_with_season_ticket(license_plate, gate_id)

- Normalize and look up the vehicle by license plate.
- Find the active RegularContract for the current time.
- Retrieve the latest open Movement (exit_time is null).
- If no open movement exists, reject the exit attempt.
- Validate the gate.
- Close the movement by setting exit_time.
- Return a structured response similar to the entry method.

These methods express the exact normal and alternative flows described in the UC2 specification and guarantee that a season ticket cannot be used by two vehicles simultaneously.

5.7.3 UC3 – Occasional Tickets (Cash Device and Exit)

The system extends the original requirements by implementing a complete workflow for occasional customers, covering entry, pricing, payment and exit.

Entry – start_occasional_entry(license_plate, gate_id)

- Atomically select a free and compatible slot:
    - No overlapping regular contracts for the current time.
    - No open occasional tickets on the same slot.
- Create an OccasionalTicket with the current entry_time and the selected slot.
- Return the slot label and ticket identifier so that the UI can display the assigned place.

Pricing – get_occasional_pricing(license_plate)

- Find the active, not-closed OccasionalTicket for the given plate.
- Calculate duration since entry_time in minutes.

- Use PricingService.get_occasional_price(...) to compute the cost based on duration and slot type.
- Store the computed amount as amount_due in the ticket.
- Return a structured result, including amount and duration_minutes, used by the cash device page.

Payment – pay_occasional_ticket(license_plate)

- Reuse get_occasional_pricing(...) to ensure amount_due is up to date.
- Use PaymentService.process_payment(...) to simulate payment processing.
- If payment fails, return an error without changing the ticket.
- If payment succeeds:
    - Set amount_paid and paid_at.
    - Compute an exit_deadline (grace period, e.g. 15 minutes after payment).
    - Save the updated ticket and return a success response including the deadline.

Exit – exit_with_occasional_ticket(license_plate, gate_id)

- Look up the active, not-closed OccasionalTicket for the license plate.
- Check if the ticket is paid (amount_paid >= amount_due).
- Verify that the current time is still within the exit_deadline (grace period).
- If the grace period has expired, reset the payment information and require a new payment.
- If everything is valid, set exit_time, mark the ticket as closed, and authorize gate opening.

5.8 Atomicity, Concurrency, and Data Integrity

To satisfy the requirements for data consistency and avoid race conditions in a multi-user environment, the implementation uses several techniques:

- transaction.atomic() around all critical operations (purchase_season_ticket, start_occasional_entry, pay_occasional_ticket, exit_with_season_ticket, exit_with_occasional_ticket)
- select_for_update() to lock:
    - Slots being reserved
    - Contracts being updated

- Occasional tickets that are in use

Additionally, guard clauses are implemented to prevent:

- Double entry with the same season ticket (open movement already exists)
- Parallel reservation of the same slot for overlapping periods
- Unpaid or expired occasional tickets from exiting

These protections ensure that UC1–UC3 behave correctly even under concurrent access and that the database state remains consistent.

5.9 Views and User Interface Implementation

Each use case is exposed to the user through Django views, which act as the presentation layer and delegate business logic to TicketService.

Season Tickets

Views:

- season_ticket_new – handles creation of season tickets
- season_ticket_list – shows the list of existing contracts for the logged-in user

Key features:

- All views are protected with @login_required
- season_ticket_new dynamically filters and displays only free slots compatible with the selected vehicle and period
- A price preview step allows users to see the calculated price before confirming the purchase
- Error messages are shown for all invalid inputs or alternative flows described in UC1

Gate UI (Seasonal + Occasional)

A unified UI style was implemented for gate interactions:

- Shared full-screen, card-like design with a background image
- Tab-like buttons to switch between Entry and Exit in the same page
- Separate paths for:
  - Season ticket gates (UC2)
  - Occasional customer gates (UC3)

The views for gate_entry, gate_exit, gate_occasional_entry and gate_occasional_exit all delegate work to TicketService and render structured messages to the user, including whether the gate is opened and why.

Cash Device

The cash device view occasional_cash_device implements the pricing and payment flow for occasional tickets:

- A simple form with a license plate field and two actions:
    - *Calculate price*
    - *Pay now*
- Displays duration, amount due, paid amount and exit deadline
- Shows success and failure messages returned by TicketService

This behavior mirrors the interaction with a real parking machine, but using a web-based UI suitable for demonstration and testing.

5.10 Testing Strategy

The project implements a comprehensive testing strategy aligned with the assignment requirements. Tests are split into two main levels: UI tests (views) and service-level tests (business logic).

5.10.1 UI Tests (Views)

UI tests are implemented using Django's TestCase and the test client. They verify that:

- Pages render correctly (status code 200)
- Forms are handled as expected
- The correct templates and messages are displayed
- Integration with the service layer is correct, while the service itself is mocked

Use-cases covered include:

- UC1: Season ticket purchase view (season_ticket_new), using a mocked TicketService
- UC2: Season gate entry and exit (gate_entry, gate_exit)
- UC3: Cash device pricing and payment (occasional_cash_device), with OccasionalTicket mocked where appropriate
- Simple smoke tests verifying that occasional entry and exit pages load successfully

This level of testing ensures that the presentation layer correctly interacts with the business logic and that the main user flows are not broken by changes in templates or form handling.

5.10.2 Service Tests (Business Logic)

Service-level tests target the TicketService class and treat it as a pure unit under test. To achieve this, all external collaborators are mocked:

- No real database access is performed
- All repositories (slot_repo, vehicle_repo, contract_repo, movement_repo, gate_repo) are MagicMock instances
- PricingService and PaymentService are also mocked

For each use-case, multiple scenarios are tested:

UC1 – Purchase Season Ticket

- Happy path: purchase succeeds when slot is free and payment is accepted
- Error: overlapping reservation is detected and purchase is rejected

UC2 – Season Ticket Entry/Exit

- Entry success when contract is active, gate exists, and no open movement exists
- Vehicle not found
- Contract not active (no valid season ticket at current time)
- Gate not found
- Exit with no open movement (ticket not in use)

UC3 – Occasional Tickets

- get_occasional_pricing: no ticket found for the plate
- get_occasional_pricing: ticket found and pricing service is called correctly
- pay_occasional_ticket: payment fails
- pay_occasional_ticket: payment succeeds and exit deadline is set
- exit_with_occasional_ticket: ticket not paid
- exit_with_occasional_ticket: grace period expired and additional payment required
- exit_with_occasional_ticket: successful exit when all conditions are met

This combination of scenarios provides near-complete coverage of the core business logic and validates that the service behaves correctly under both normal and error conditions.

5.10.3 Summary of the Implementation

The PortugueseParking implementation reflects a clean architecture that separates:

- UI (Django views and templates)
- Business rules (TicketService and auxiliary services)
- Persistence (Django ORM models in each app)
- Integration concerns (pricing and payment services)

The system fully implements the required functionality for:

- UC1 – Purchasing season tickets
- UC2 – Entering/exiting with season tickets
- UC3 – Managing occasional tickets, including entry, pricing, payment, and exit with grace period enforcement

In addition, the project delivers:

- A structured, modular folder hierarchy
- Dependency-injected service objects for easier testing
- Comprehensive unit and view-level tests with proper isolation
- Transactional safety and concurrency protection for critical operations

Together, these elements form a robust prototype of a parking management system that is both faithful to the original requirements and extensible for future enhancements.

6 Conclusion

The implementation of the PortugueseParking system successfully transforms the conceptual architecture and requirements into a fully functional Django application. Across the three core use cases, season ticket purchase (UC1), season ticket entry/exit (UC2), and occasional parking with pricing and payment (UC3) the system demonstrates a clean separation between UI, business logic, and persistence, reflecting the layered architecture defined earlier in the project.

Through the use of atomic transactions, concurrency-safe operations, and comprehensive validation rules, the application ensures data integrity and correct behavior under realistic multi-user conditions. The TicketService consolidates the domain logic and provides a robust foundation for future extensions, such as integrating real payment APIs or physical gate hardware. The user interface, built with Django views and templates, mirrors real parking workflows while remaining simple and intuitive for demonstration purposes.

A strong testing strategy including both UI-level tests with mocks and pure service-level unit tests ensures reliability, correctness, and maintainability. The test suite covers normal and alternative flows, error handling, and edge conditions, providing confidence that the system behaves as expected under varied scenarios.

Overall, the project delivers a solid prototype aligned with the functional and non-functional requirements, providing a realistic blueprint for a scalable, modern parking management solution. The modular architecture and comprehensive implementation offer a strong foundation for further development in subsequent project stages.