Design a Parking Lot

Let's make an object-oriented design for a multi-floor Parking Lot

We'll cover the following:

A parking lot or car park is a dedicated cleared area that is intended for parking vehicles. In most countries where cars are a major mode of transportation, parking lots are a feature of every city and suburban area. Shopping malls, sports stadiums, megachurches, and similar venues often feature parking lots over large areas.

Parking Lot

System Requirements

We will focus on the following set of requirements while designing the parking lot:

1. The parking lot should have multiple floors where customers can park their cars.

2. The parking lot should have multiple entry and exit points.
3. Customers can collect a parking ticket from the entry points and can pay the parking fee at the exit points on their way out.
4. Customers can pay the tickets at the automated exit panel or to the parking attendant.
5. Customers can pay via both cash and credit cards.
6. Customers should also be able to pay the parking fee at the customer's info portal on each floor. If the customer has paid at the info portal, they don't have to pay at the exit.
7. The system should not allow more vehicles than the maximum capacity of the parking lot. If the parking is full, the system should be able to show a message at the entrance panel and on the parking display board on the ground floor.
8. Each parking floor will have many parking spots. The system should support multiple types of parking spots such as Compact, Large, Handicapped, Motorcycle, etc.
9. The Parking lot should have some parking spots specified for electric cars. These spots should have an electric panel through which customers can pay and charge their vehicles.
10. The system should support parking for different types of vehicles like car, truck, van, motorcycle, etc.
11. Each parking floor should have a display board showing any free parking spot for each spot type.
12. The system should support a per-hour parking fee model. For example, customers have to pay $4 for the first hour$,3.5 for the second and third hours, and $2.5 for all the remaining hours.

Use Case Diagram

Here are the main Actors in our system:

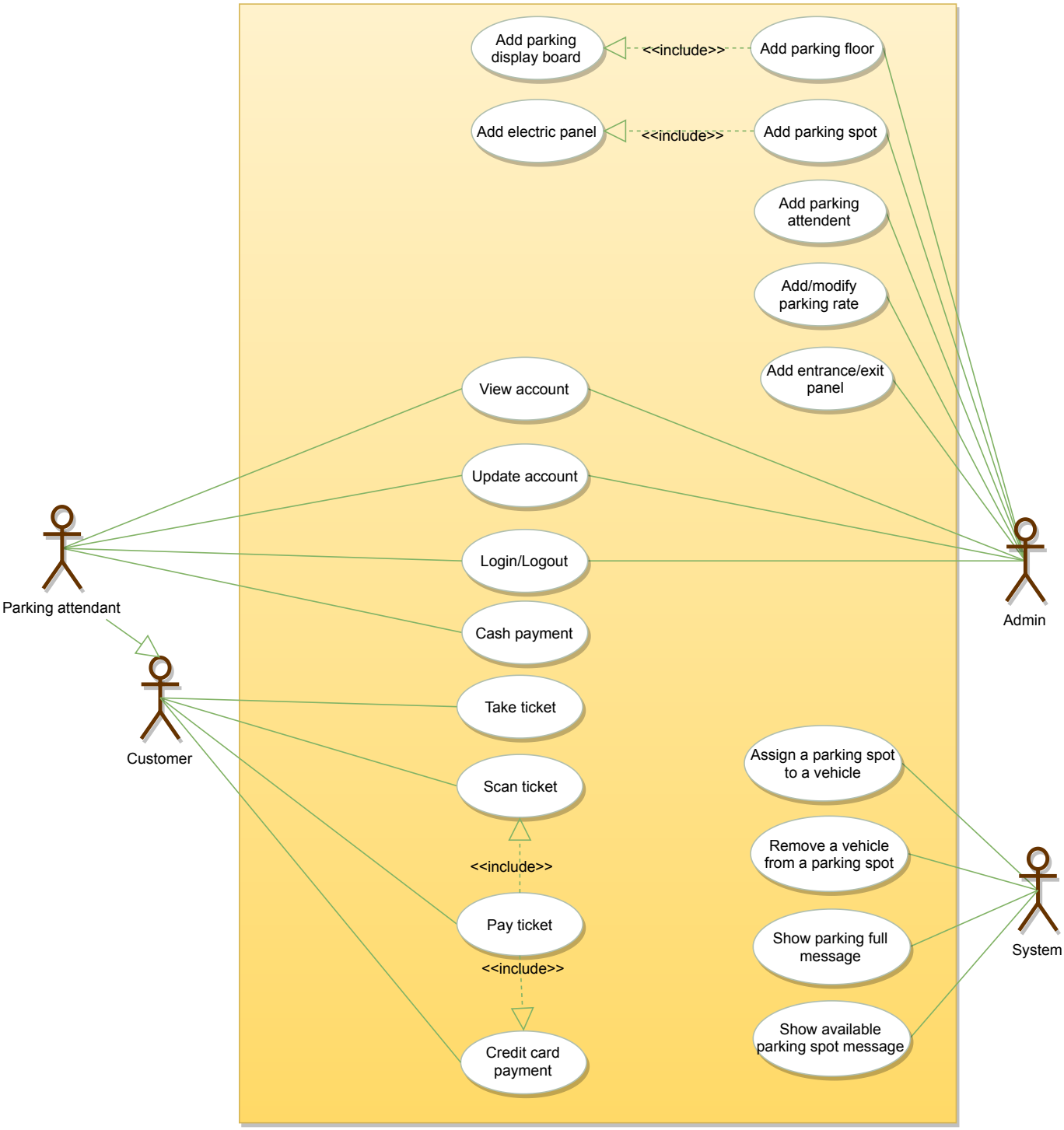
- **Admin:** Mainly responsible for adding and modifying parking floors, parking spots, entrance, and exit panels, adding/removing parking attendants, etc.
- **Customer:** All customers can get a parking ticket and pay for it.
- **Parking Attendant:** Parking attendants can do all the activities on the customer's behalf, and can take cash for ticket payment.
- **System:** To display messages on different info panels, as well as assigning and removing a vehicle from a parking spot.

Here are the top use cases for Parking Lot:

- **Add/Remove/Edit parking floor:** To add, remove or modify a parking floor from the system. Each floor can have its own display board to show free parking spots.

- **Add/Remove/Edit parking spot:** To add, remove or modify a parking spot on a parking floor.
- **Add/Remove a parking attendant:** To add or remove a parking attendant from the system.
- **Take ticket:** To provide customers with a new parking ticket when entering the parking lot.
- **Scan ticket:** To scan a ticket to find out the total charge.
- **Credit card payment:** To pay the ticket fee with credit card.
- **Cash payment:** To pay the parking ticket through cash.
- **Add/Modify parking rate:** To allow admin to add or modify the hourly parking rate.

Here is the use case diagram of our Parking Lot:



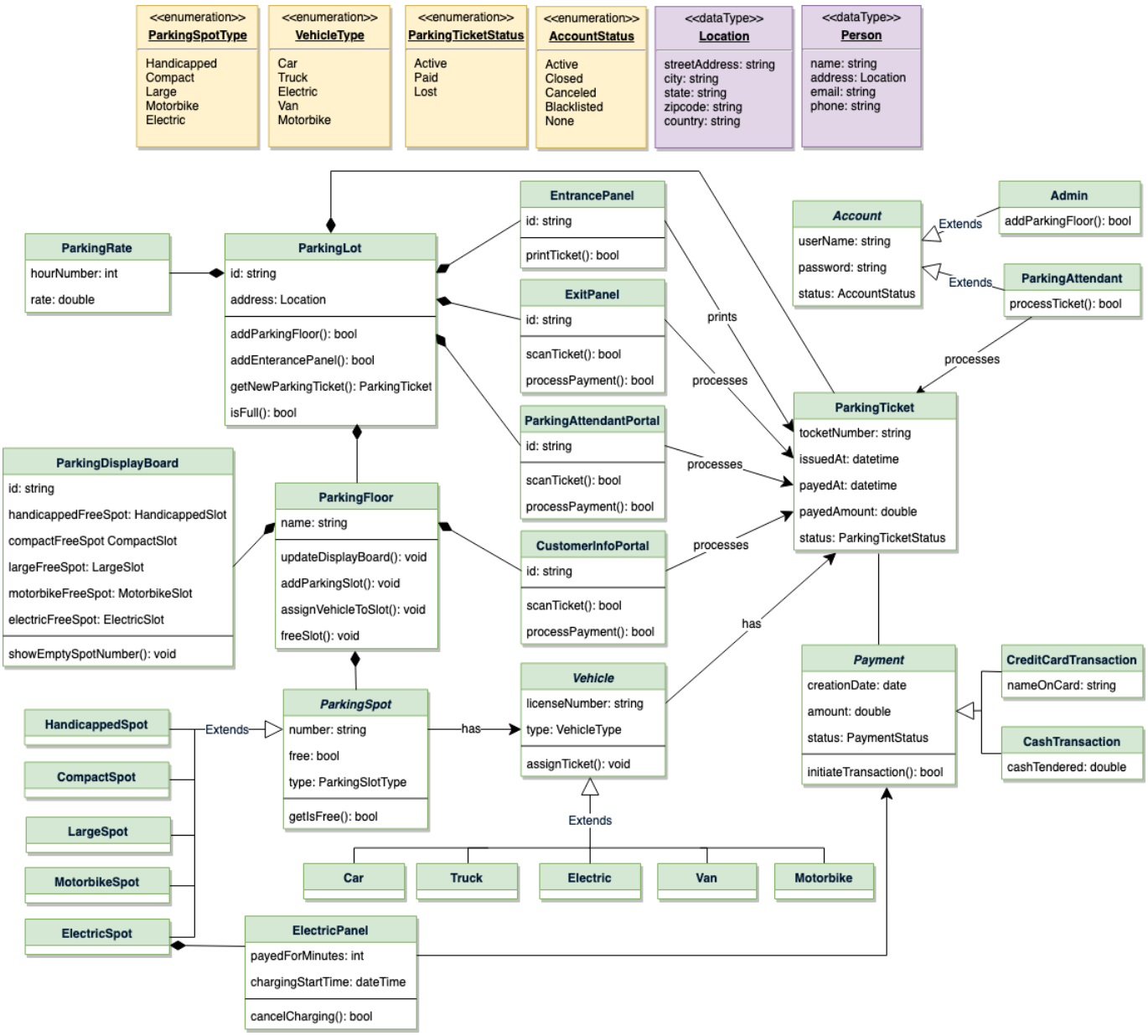
Use Case Diagram for Parking Lot

Class Diagram

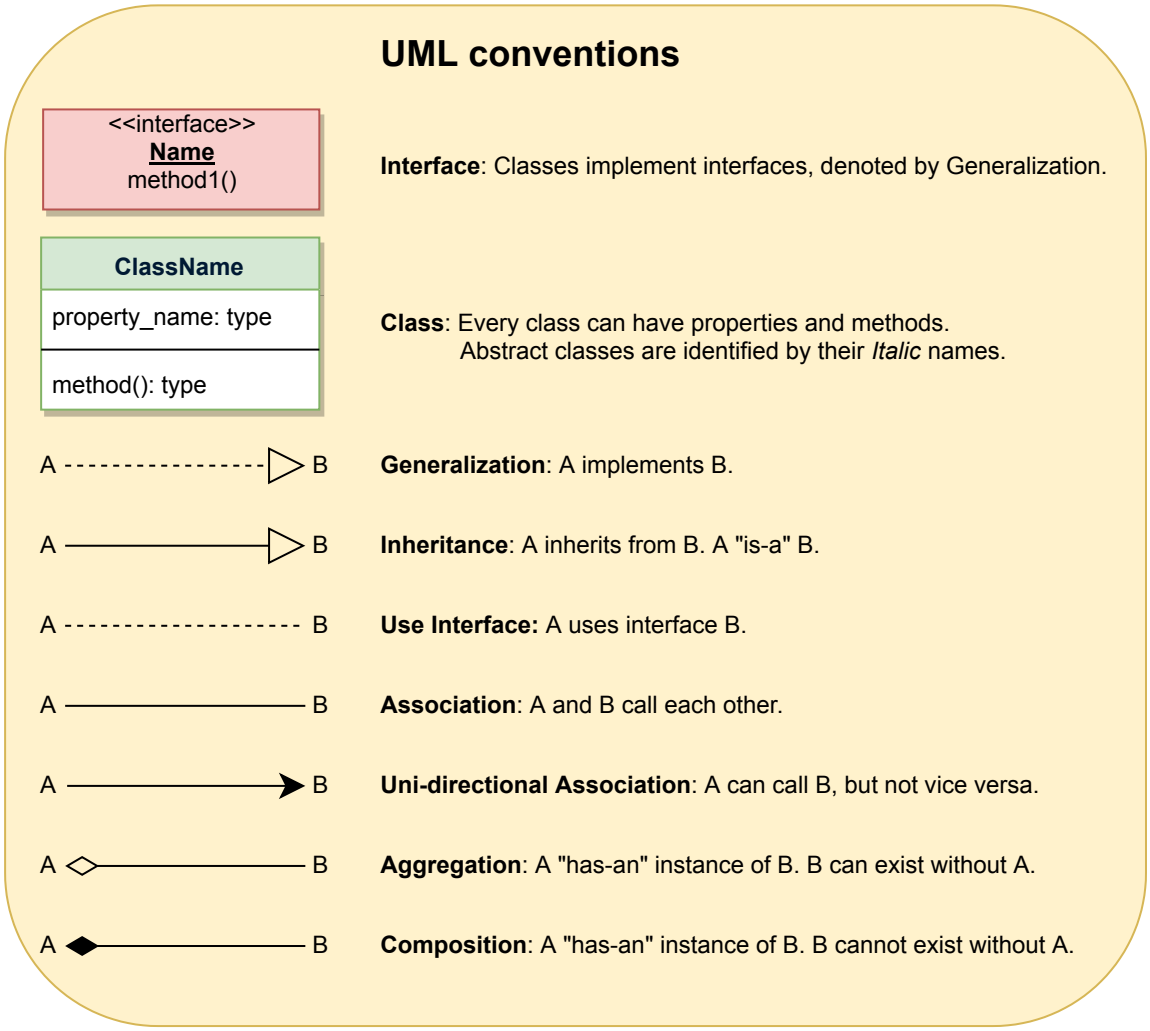
Here are the main classes of our Parking Lot System:

- **ParkingLot:** The central part of the organization for which this software has been designed. It has attributes like ‘Name’ to distinguish it from any other parking lots and ‘Address’ to define its location.
- **ParkingFloor:** The parking lot will have many parking floors.

- **ParkingSpot:** Each parking floor will have many parking spots. Our system will support different parking spots 1) Handicapped, 2) Compact, 3) Large, 4) Motorcycle, and 5) Electric.
- **Account:** We will have two types of accounts in the system: one for an Admin, and the other for a parking attendant.
- **Parking ticket:** This class will encapsulate a parking ticket. Customers will take a ticket when they enter the parking lot.
- **Vehicle:** Vehicles will be parked in the parking spots. Our system will support different types of vehicles 1) Car, 2) Truck, 3) Electric, 4) Van and 5) Motorcycle.
- **EntrancePanel and ExitPanel:** EntrancePanel will print tickets, and ExitPanel will facilitate payment of the ticket fee.
- **Payment:** This class will be responsible for making payments. The system will support credit card and cash transactions.
- **ParkingRate:** This class will keep track of the hourly parking rates. It will specify a dollar amount for each hour. For example, for a two hour parking ticket, this class will define the cost for the first and the second hour.
- **ParkingDisplayBoard:** Each parking floor will have a display board to show available parking spots for each spot type. This class will be responsible for displaying the latest availability of free parking spots to the customers.
- **ParkingAttendantPortal:** This class will encapsulate all the operations that an attendant can perform, like scanning tickets and processing payments.
- **CustomerInfoPortal:** This class will encapsulate the info portal that customers use to pay for the parking ticket. Once paid, the info portal will update the ticket to keep track of the payment.
- **ElectricPanel:** Customers will use the electric panels to pay and charge their electric vehicles.



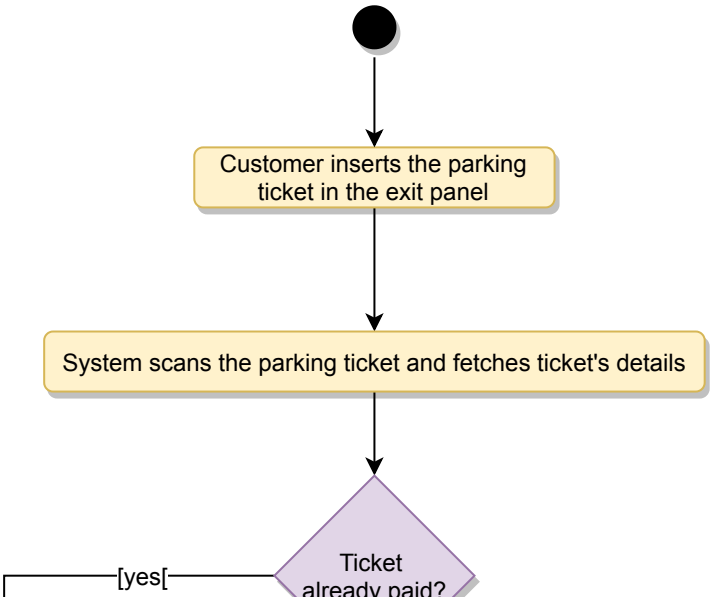
Class Diagram for Parking Lot

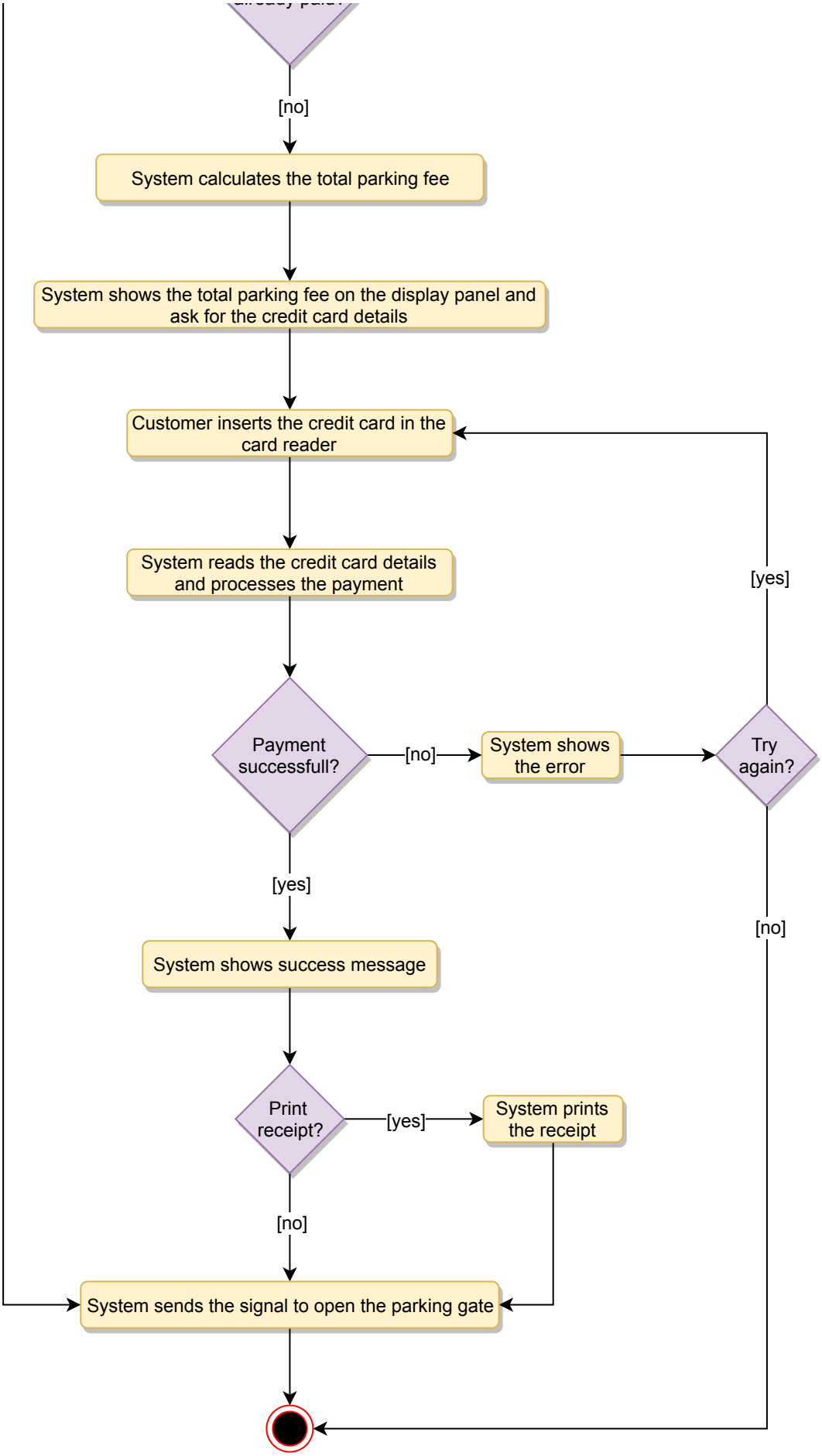


UML for Parking Lot

Activity Diagram

Customer paying for parking ticket: Any customer can perform this activity. Here are the set of steps:





Activity Diagram for Parking Lot Parking Ticket

Code

Following is the skeleton code for our parking lot system:

Enums and Constants: Here are the required enums, data types, and constants:

```
from enum import Enum


class VehicleType(Enum):
    CAR, TRUCK, ELECTRIC, VAN, MOTORBIKE = 1, 2, 3, 4, 5


class ParkingSpotType(Enum):
    HANDICAPPED, COMPACT, LARGE, MOTORBIKE, ELECTRIC = 1, 2, 3, 4, 5


class AccountStatus(Enum):
    ACTIVE, BLOCKED, BANNED, COMPROMISED, ARCHIVED, UNKNOWN = 1, 2, 3, 4, 5, 6


class ParkingTicketStatus(Enum):
    ACTIVE, PAID, LOST = 1, 2, 3


class Address:
    def __init__(self, street, city, state, zip_code, country):
        self.__street_address = street
        self.__city = city
        self.__state = state
        self.__zip_code = zip_code
        self.__country = country


class Person():
    def __init__(self, name, address, email, phone):
        self.__name = name
        self.__address = address
        self.__email = email
        self.__phone = phone
```

Account, Admin, and ParkingAttendant: These classes represent various people that interact with our system:

```python
from .constants import *


class Account:
    def __init__(self, user_name, password, person,
status=AccountStatus.Active):
        self.__user_name = user_name
        self.__password = password
        self.__person = person
        self.__status = status

    def reset_password(self):
        None


class Admin(Account):
    def __init__(self, user_name, password, person,
status=AccountStatus.Active):
        super().__init__(user_name, password, person, status)

    def add_parking_floor(self, floor):
        None

    def add_parking_spot(self, floor_name, spot):
        None

    def add_parking_display_board(self, floor_name, display_board):
        None

    def add_customer_info_panel(self, floor_name, info_panel):
        None

    def add_entrance_panel(self, entrance_panel):
        None

    def add_exit_panel(self, exit_panel):
        None


class ParkingAttendant(Account):
    def __init__(self, user_name, password, person,
status=AccountStatus.Active):
        super().__init__(user_name, password, person, status)

    def process_ticket(self, ticket_number):
        None
```

ParkingSpot: Here is the definition of ParkingSpot and all of its children classes:

```python
from abc import ABC
from .constants import  *


class ParkingSpot(ABC):
    def __init__(self, number, parking_spot_type):
        self.__number = number
        self.__free = True
        self.__vehicle = None
        self.__parking_spot_type = parking_spot_type

    def is_free(self):
        return self.__free

    def assign_vehicle(self, vehicle):
        self.__vehicle = vehicle
        self.__free = False

    def remove_vehicle(self):
        self.__vehicle = None
        self.free = True


class HandicappedSpot(ParkingSpot):
    def __init__(self, number):
        super().__init__(number, ParkingSpotType.HANDICAPPED)


class CompactSpot(ParkingSpot):
    def __init__(self, number):
        super().__init__(number, ParkingSpotType.COMPACT)


class LargeSpot(ParkingSpot):
    def __init__(self, number):
        super().__init__(number, ParkingSpotType.LARGE)


class MotorbikeSpot(ParkingSpot):
    def __init__(self, number):
        super().__init__(number, ParkingSpotType.MOTORBIKE)


class ElectricSpot(ParkingSpot):
    def __init__(self, number):
        super().__init__(number, ParkingSpotType.ELECTRIC)
```

ParkingFloor: This class encapsulates a parking floor:

```
from .constants import  *
from .parking_display_board import *


class ParkingFloor:
    def __init__(self, name):
        self.__name = name
        self.__handicapped_spots = {}
        self.__compact_spots = {}
        self.__large_spots = {}
        self.__motorbike_spots = {}
        self.__electric_spots = {}
        self.__info_portals = {}
        self.__free_handicapped_spot_count = {'free_spot': 0}
        self.__free_compact_spot_count = {'free_spot': 0}
        self.__free_large_spot_count = {'free_spot': 0}
        self.__free_motorbike_spot_count = {'free_spot': 0}
        self.__free_electric_spot_count = {'free_spot': 0}
        self.__display_board = ParkingDisplayBoard()

    def add_parking_spot(self, spot):
        switcher = {
            ParkingSpotType.HANDICAPPED:
self.__handicapped_spots.put(spot.get_number(), spot),
            ParkingSpotType.COMPACT:
self.__compact_spots.put(spot.get_number(), spot),
            ParkingSpotType.LARGE: self.__large_spots.put(spot.get_number(),
spot),
            ParkingSpotType.MOTORBIKE:
self.__motorbike_spots.put(spot.get_number(), spot),
            ParkingSpotType.ELECTRIC:
self.__electric_spots.put(spot.get_number(), spot),
        }
        switcher.get(spot.get_type(), 'Wrong parking spot type')

    def assign_vehicleToSpot(self, vehicle, spot):
        spot.assign_vehicle(vehicle)
        switcher = {
            ParkingSpotType.HANDICAPPED:
self.update_display_board_for_handicapped(spot),
            ParkingSpotType.COMPACT:
self.update_display_board_for_compact(spot),
            ParkingSpotType.LARGE: self.update_display_board_for_large(spot),
            ParkingSpotType.MOTORBIKE:
self.update_display_board_for_motorbike(spot),
            ParkingSpotType.ELECTRIC:
self.update_display_board_for_electric(spot),
```

```
        }
        switcher(spot.get_type(), 'Wrong parking spot type!')

    def update_display_board_for_handicapped(self, spot):
        if self.__display_board.get_handicapped_free_spot().get_number() ==
spot.get_number():
            # find another free handicapped parking and assign to display_board
            for key in self.__handicapped_spots:
                if self.__handicapped_spots.get(key).is_free():

self.__display_board.set_handicapped_free_spot(self.__handicapped_spots.get(key
))

            self.__display_board.show_empty_spot_number()

    def update_display_board_for_compact(self, spot):
        if self.__display_board.get_compact_free_spot().get_number() ==
spot.get_number():
            # find another free compact parking and assign to display_board
            for key in self.__compact_spots.key_set():
                if self.__compact_spots.get(key).is_free():

self.__display_board.set_compact_free_spot(self.__compact_spots.get(key))

            self.__display_board.show_empty_spot_number()

    def free_spot(self, spot):
        spot.remove_vehicle()
        switcher = {
            ParkingSpotType.HANDICAPPED:
self.__free_handicapped_spot_count.update(
              free_spot = self.__free_handicapped_spot_count["free_spot"] + 1
            ),
            ParkingSpotType.COMPACT: self.__free_compact_spot_count.update(
              free_spot=self.__free_compact_spot_count["free_spot"] + 1
            ),
            ParkingSpotType.LARGE: self.__free_large_spot_count.update(
              free_spot=self.__free_large_spot_count["free_spot"] + 1
            ),
            ParkingSpotType.MOTORBIKE: self.__free_motorbike_spot_count.update(
              free_spot=self.__free_motorbike_spot_count["free_spot"] + 1
            ),
            ParkingSpotType.ELECTRIC: self.__free_electric_spot_count.update(
              free_spot=self.__free_electric_spot_count["free_spot"] + 1
            ),
        }

        switcher(spot.get_type(), 'Wrong parking spot type!')
```

ParkingDisplayBoard: This class encapsulates a parking display board:

```
class ParkingDisplayBoard:
    def __init__(self, id):
        self.__id = id
        self.__handicapped_free_spot = None
        self.__compact_free_spot = None
        self.__large_free_spot = None
        self.__motorbike_free_spot = None
        self.__electric_free_spot = None

    def show_empty_spot_number(self):
        message = ""
        if self.__handicapped_free_spot.is_free():
            message += "Free Handicapped: " +
self.__handicapped_free_spot.get_number()
        else:
            message += "Handicapped is full"
        message += "\n"

        if self.__compact_free_spot.is_free():
            message += "Free Compact: " + self.__compact_free_spot.get_number()
        else:
            message += "Compact is full"
        message += "\n"

        if self.__large_free_spot.is_free():
            message += "Free Large: " + self.__large_free_spot.get_number()
        else:
            message += "Large is full"
        message += "\n"

        if self.__motorbike_free_spot.is_free():
            message += "Free Motorbike: " +
self.__motorbike_free_spot.get_number()
        else:
            message += "Motorbike is full"
        message += "\n"

        if self.__electric_free_spot.is_free():
            message += "Free Electric: " +
self.__electric_free_spot.get_number()
        else:
            message += "Electric is full"

        print(message)
```

ParkingLot: Our system will have only one object of this class. This can be enforced by using the Singleton pattern. In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to only one object.

```
import threading
from .constants import *


class ParkingLot:
    # singleton ParkingLot to ensure only one object of ParkingLot in the
system,
    # all entrance panels will use this object to create new parking ticket:
get_new_parking_ticket(),
    # similarly exit panels will also use this object to close parking tickets
    instance = None

    class __OnlyOne:
        def __init__(self, name, address):
        # 1. initialize variables: read name, address and parking_rate from
database
        # 2. initialize parking floors: read the parking floor map from
database,
        #    this map should tell how many parking spots are there on each
floor. This
        #    should also initialize max spot counts too.
        # 3. initialize parking spot counts by reading all active tickets from
database
        # 4. initialize entrance and exit panels: read from database

            self.__name = name
            self.__address = address
            self.__parking_rate = ParkingRate()

            self.__compact_spot_count = 0
            self.__large_spot_count = 0
            self.__motorbike_spot_count = 0
            self.__electric_spot_count = 0
            self.__max_compact_count = 0
            self.__max_large_count = 0
            self.__max_motorbike_count = 0
            self.__max_electric_count = 0

            self.__entrance_panels = {}
            self.__exit_panels = {}
            self.__parking_floors = {}

            # all active parking tickets, identified by their ticket_number
            self.__active_tickets = {}

            self.__lock = threading.Lock()
```

```
    def __init__(self, name, address):
        if not ParkingLot.instance:
            ParkingLot.instance = ParkingLot.__OnlyOne(name, address)
        else:
            ParkingLot.instance.__name = name
            ParkingLot.instance.__address = address

    def __getattr__(self, name):
        return getattr(self.instance, name)

    def get_new_parking_ticket(self, vehicle):
        if self.is_full(vehicle.get_type()):
            raise Exception('Parking full!')
    # synchronizing to allow multiple entrances panels to issue a new
    # parking ticket without interfering with each other
        self.__lock.acquire()
        ticket = ParkingTicket()
        vehicle.assign_ticket(ticket)
        ticket.save_in_DB()
        # if the ticket is successfully saved in the database, we can increment
the parking spot count
        self.__increment_spot_count(vehicle.get_type())
        self.__active_tickets.put(ticket.get_ticket_number(), ticket)
        self.__lock.release()
        return ticket

    def is_full(self, type):
        # trucks and vans can only be parked in LargeSpot
        if type == VehicleType.Truck or type == VehicleType.Van:
            return self.__large_spot_count >= self.__max_large_count

        # motorbikes can only be parked at motorbike spots
        if type == VehicleType.Motorbike:
            return self.__motorbike_spot_count >= self.__max_motorbike_count

        # cars can be parked at compact or large spots
        if type == VehicleType.Car:
            return (self.__compact_spot_count + self.__large_spot_count) >=
(self.__max_compact_count + self.__max_large_count)

        # electric car can be parked at compact, large or electric spots
        return (self.__compact_spot_count + self.__large_spot_count +
self.__electric_spot_count) >= (self.__max_compact_count +
self.__max_large_count

+ self.__max_electric_count)

    # increment the parking spot count based on the vehicle type
```

```
    def increment_spot_count(self, type):
        large_spot_count = 0
        motorbike_spot_count = 0
        compact_spot_count = 0
        electric_spot_count = 0
        if type == VehicleType.Truck or type == VehicleType.Van:
            large_spot_count += 1
        elif type == VehicleType.Motorbike:
            motorbike_spot_count += 1
        elif type == VehicleType.Car:
            if self.__compact_spot_count < self.__max_compact_count:
                compact_spot_count += 1
            else:
                large_spot_count += 1
        else:  # electric car
            if self.__electric_spot_count < self.__max_electric_count:
                electric_spot_count += 1
            elif self.__compact_spot_count < self.__max_compact_count:
                compact_spot_count += 1
            else:
                large_spot_count += 1

        def is_full(self):
            for key in self.__parking_floors:
                if not self.__parking_floors.get(key).is_full():
                    return False
            return True

        def add_parking_floor(self, floor):
            # store in database
            None

        def add_entrance_panel(self, entrance_panel):
            # store in database
            None

        def add_exit_panel(self,  exit_panel):
            # store in database
            None
```