Parking-API

Architecture

Parking-API is a RESTful web services written in Java. This project uses the following technologies:

- Spring Boot;
- Database H2 (Relational DBMS);
- Maven.

This document is intended to give an overview of the architecture of Parking-API.

The design of Parking-API followed the most common guidelines and design patterns, such as SOLID principles and MVC.

The project is composed by different layers:

- **Data layers**: here resides the database;
- **Data Access Objects**: this layer keeps the application's domain model completely agnostic about the database. DAO provides interface to access to the database.
- **Service**: the business logic is implemented by the Service components.
- **REST controllers**: they represent the endpoints which are used by the client to communicate to Parking-API.

This document focuses more on business logic. For this reason, Service layer and Domain model are deeply explained.

Service layer

It is the heart of the web service, where the business logic is implemented. The service components communicate with the data layer through the interface provided by the DAO. They exchange data with the REST controllers as well. In Service layer we can recognize different type of classes:

ManagerService class

These components have been implemented to provide most commons CRUD operations for a specific domain object. There are three **ManagerService**:

- **ParkingManagerService**: it is responsible to handle the domain objects related to Parking, parking slots and parking access;
- **VehicleManagerService**: it works mostly with Vehicle model;
- **PricingManagerService**: it oversees the operations related to the pricing policies.

The *ManagerService* can communicate among each other to get data they need to execute their logic.

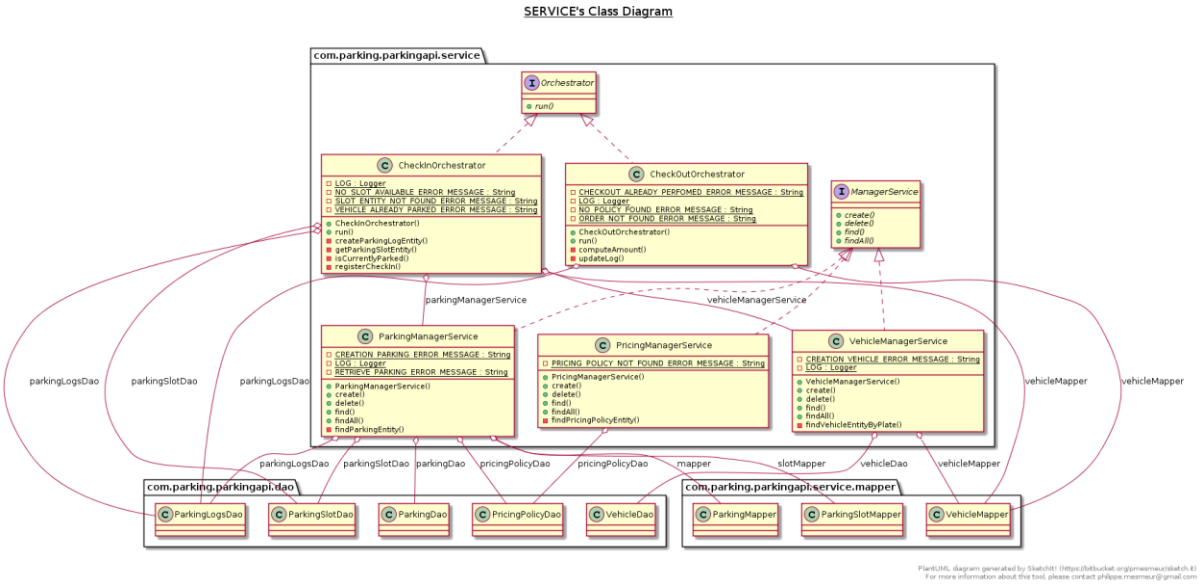
They are helped by other classes, called Mappers, which are responsible to perform mapping between JPA entity and Domain objects, and vice-versa.

Orchestrator

Orchestrators are service components like ManagerService. The biggest difference is that Orchestrators are process oriented, while ManagerService are more Domain objects oriented.

There are two types of Orchestrators, one for **Check-in** and another one for **Check-out**.

Orchestrators invoke ManagerService and DAOs to achieve their purposes.



Domain Model

Pricing Policy model

The classes related to pricing policy have been implemented through the *decorator pattern*. Since one of the requirements indicates that in future new pricing policy can be adopted, decorator pattern provides enough flexibility by composition. Furthermore, it can avoid the explosion of number of new classes in future.

All the pricing policy implements the **PricingPolicy** interface.

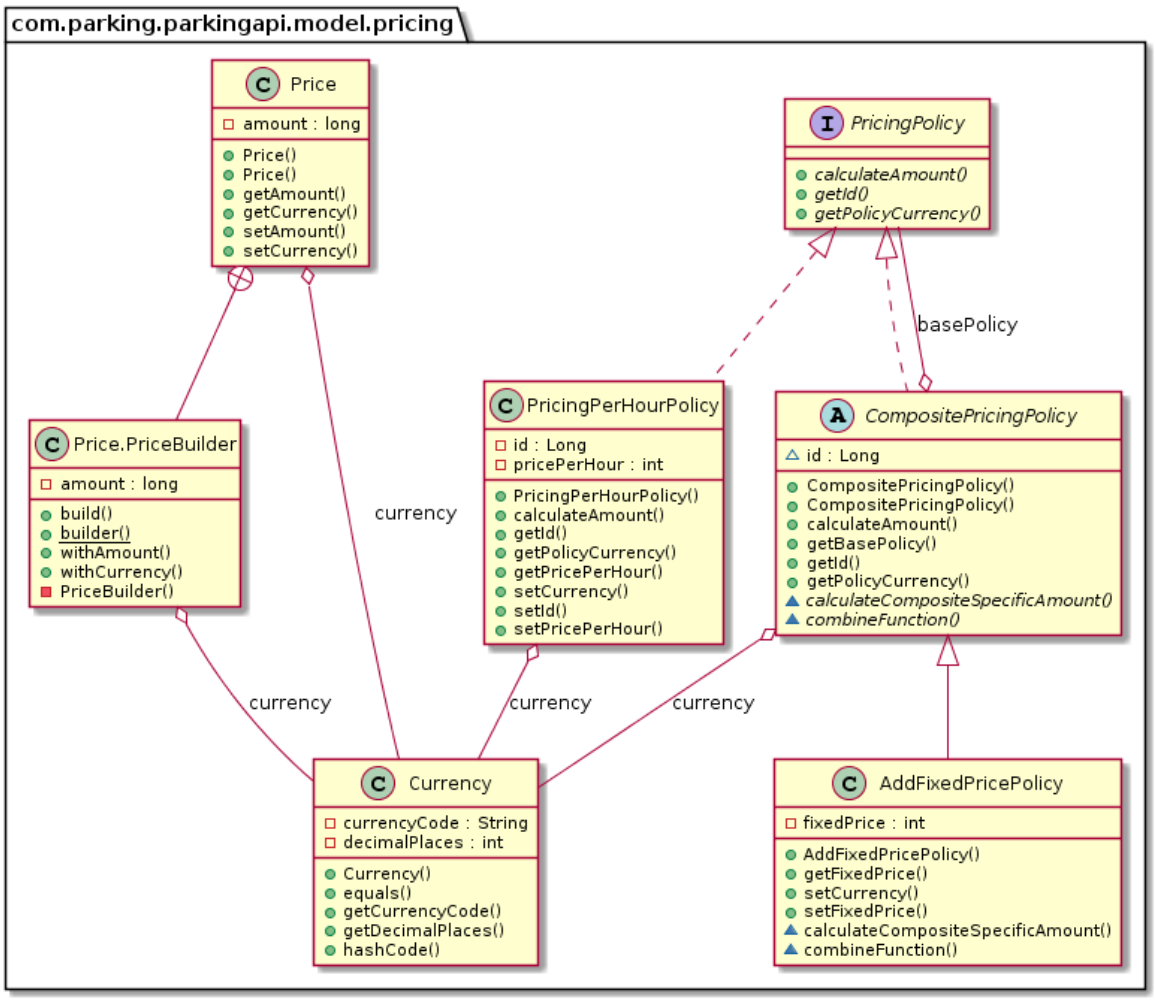
Then we can have two type of policies:

- **Base** or simple policy;
- **Composite** policy.

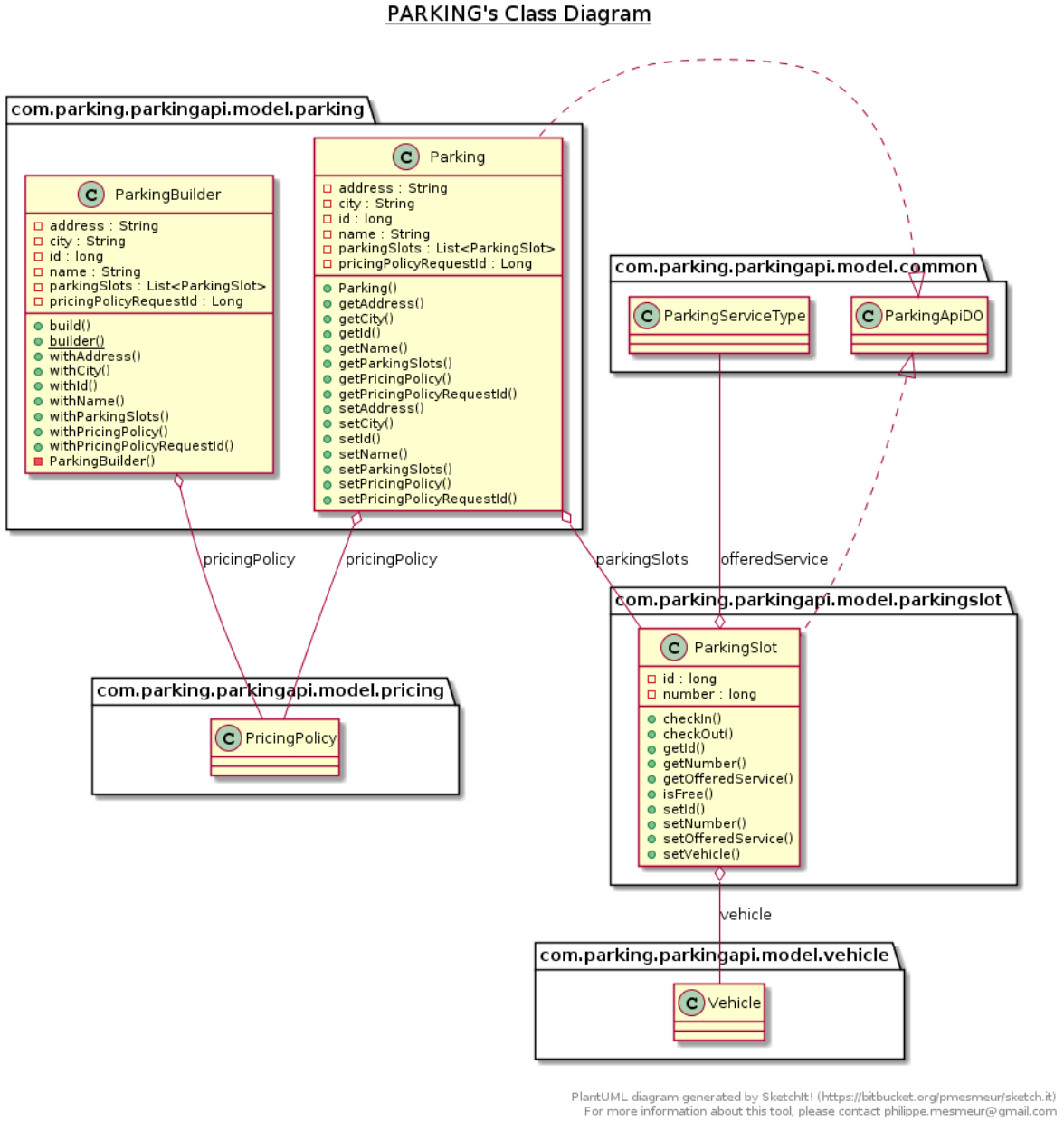
A composite policy applies its own conditions on top of another pricing policy.

In the case of Parking-API we have **AddFixedPricePolicy** which adds a fixed amount to the price computed by the base policy **PricePerHourPolicy.**

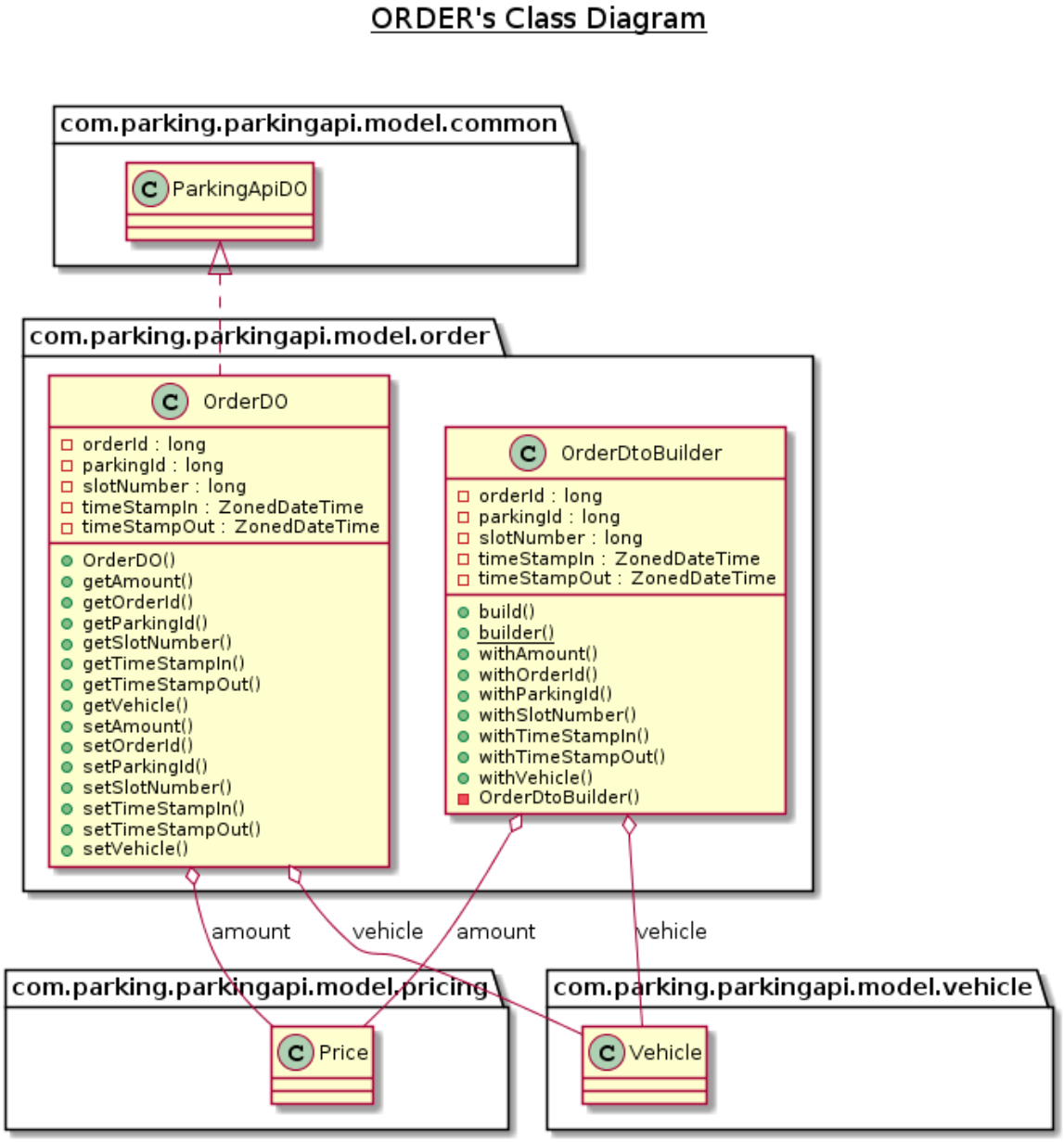
PRICING's Class Diagram



Parking model



Order model


ORDER's Class Diagram
com.parking.parkingapi.model.common
ParkingApiDO
com.parking.parkingapi.model.order
OrderDO
orderId : long
parkingId : long
slotNumber : long
timeStampIn : ZonedDateTime
timeStampOut : ZonedDateTime
OrderDO()
getAmount()
getOrderId()
getParkingId()
getSlotNumber()
getTimeStampIn()
getTimeStampOut()
getVehicle()
setAmount()
setOrderId()
setParkingId()
setSlotNumber()
setTimeStampIn()
setTimeStampOut()
setVehicle()
OrderDtoBuilder
orderId : long
parkingId : long
slotNumber : long
timeStampIn : ZonedDateTime
timeStampOut : ZonedDateTime
build()
builder()
withAmount()
withOrderId()
withParkingId()
withSlotNumber()
withTimeStampIn()
withTimeStampOut()
withVehicle()
OrderDtoBuilder()
amount
vehicle
amount
vehicle
com.parking.parkingapi.model.pricing
Price
com.parking.parkingapi.model.vehicle
Vehicle