Software Engineering 2 Project

Design Document

PowerEnJoy

POLITECNICO DI MILANO

Team: Nico Montali, Enrico Fini

Contents

Document Status

Document Title	Design Document
Document ID	PowerEnjoy/DD/1.11
Authors	N. Montali, E. Fini
Version	1.11

Changelog

Version	**Date**	**Changes**
1.0	11/12/2016	Initial version
1.1	04/01/2017	• Added document status • Fixed some typos • Added missing interface link between RouterVehicleInt and VehicleManager • Added hours of work based on log file
1.11	09/01/2017	• Changed the architecture of the Vehicle-Core interaction to only vehicle as a server • Removed RouterVehicleInt since vehicle is now treated as a server only • Extended the description of the Router due to redesign of the component • Added new parameters to Router in the Interface Diagram • Updated paragraph 2.2.1 with car APIs

Introduction

1.1 Purpose

The purpose of this document is to give a more detailed and technical overview about PowerEnJoy system.
This document is addressed to developers and aims to identify:

- The high level architecture;
- The design patterns;
- The main components and their interfaces provided one for another;
- The Runtime behaviour.

1.2 Scope

This document will present different level views in order to describe clearly the architecture of PowerEnJoy System. In particular we will present the component view, both high and low level, the runtime view and a further description of user interface, analysed in its runtime flow.
The System is based on two mobile applications and a web application.
As already explained in the RASD, the targets for the Service are:

- The Drivers;
- The Workers;
- The Admins.

The System allows the Drivers to reserve cars from both the mobile application and the web portal. At the same time it manages the the status of the cars and, if necessary, assigns tasks to the Workers via an ad-hoc mobile application. The Admins of the System can interact with the web portal to make privileged actions.

1.3 Definitions, Acronyms and Abbreviations

Driver Client of the system, i.e. the one that uses the service, reserves and drives. Every Driver must provide personal informations (name, surname, email, birthdate, a valid license and payment information;

Ride A single usage of the service, starts when the user turn on the engine, stops when the car is locked inside a Safe Area;

Blocked driver A driver that has an invalid license or invalid payment coordinates;

Worker Employee of the company, perform physical actions (moving, charging etc) on cars;

Report Car issue reported by the Driver during a Drive. Will later be assigned to a Worker;

Task Piece of work assigned to a worker. Different type of tasks are described later. Every task assigns a single car to a single worker;

Administrator Employee of PowerEni that is in charge of the administration of the system through the administration console;

Drop off The act of leaving the car inside a safe area. This ends the service provided to the driver and effectively make the driver pay;

Safe Area An area in which the Driver can park the car and stop the Service;

RASD Requirements Analysis and Specifications Document;

DD Design Document;

API Application Programming Interface;

DBMS DataBase Management System;

UI User Interface;

REST REpresantional State Transfer.

1.4 Reference Documents

- The RASD we released before;
- Sample Design Deliverable Discussed on Nov. 2.pdf.

1.5 Document Structure

Introduction: contains an overall description of the content and the structure of the document;

Architecture: this section contains:

- **Overview:** coarse view of the System i.e. the division in tiers;
- **High Level Components:** description of the main components of the application and their interaction;
- **Tracking Vehicle State:** more detailed description of the vehicle tracking mechanism;
- **Core Component:** describes the inner composition of the core component;
- **Relevant State Charts:** gives a deeper explanation of the dynamic behaviour of the model using State Charts;
- **Runtime View:** contains some Sequence Diagrams of the main actions of the System;
- **Interfaces:** contains the interfaces of all the subcomponents of the Core Component.

Algorithm Design: we will present some algorithms with pseudocode to show the logic of the most important parts of the System;

User Interface Design: since the funcionalities contained in this Document and in the RASD are very explanatory for themselves, we do not include any further expansions of the UI design of the RASD.

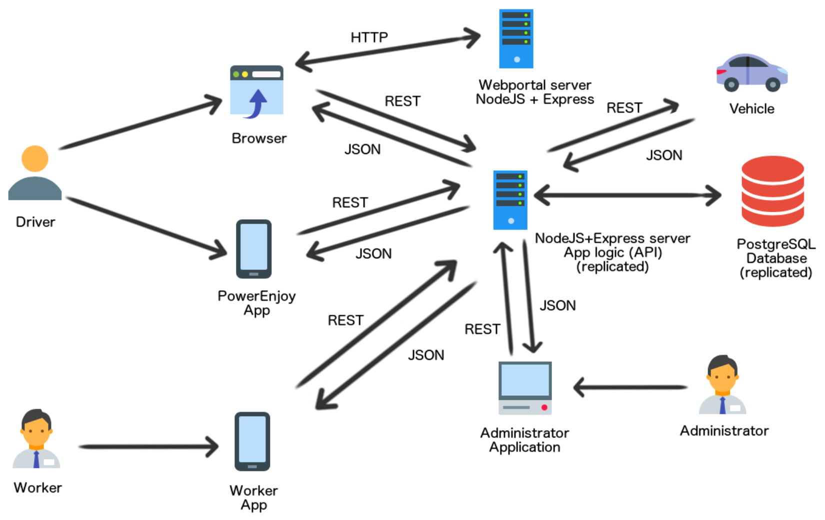
Requirements Traceability: An explanation of how our architectural choices respect the requirements described in the RASD.

Architecture

2.1 Overview

The PowerEnjoy system is designed as a 3-tier application:

- **Data tier:** A PostgreSQL DB, deployed on AWS and replicated for availability and integrity.
- **Application tier:** A NodeJS application that provides APIs, deployed on AWS and replicated. The Express library is used to realise API endpoints. These APIs are exposed in a REST manner, responses are JSONs. The replication of the application grants fault tolerance and, using a load balancer, also availability.
- **Presentation tier:** The presentation layer is realised with the 2 mobile applications (Driver and Worker), the web portal (Driver) and the Administrator application. The web portal is realised using EJS templating (server-side) and JQuery (client-side).

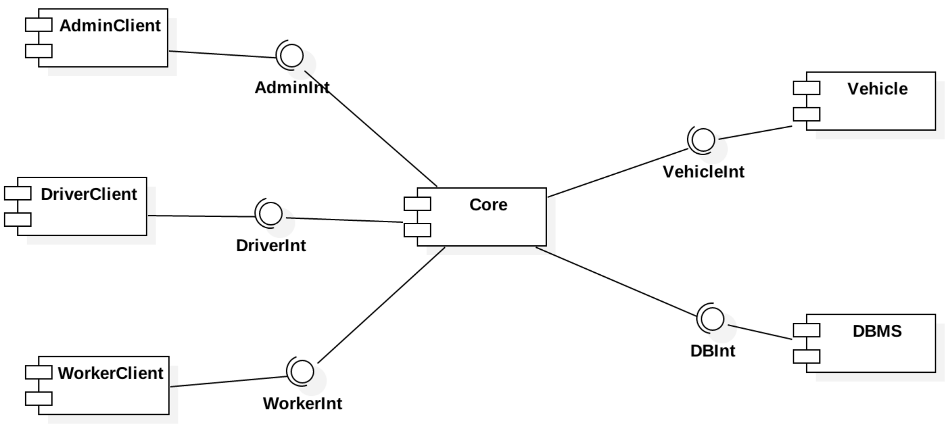


2.2 High level components

The high level structure of the System can be divided into 6 components. The main component is the **core**, that receives requests from **clients**, either **drivers**, **workers** and **administrators**; depending on the client type, the component provides different interfaces. The core also communicates with the **DBMS** component, responsible of data persistence. The core is also responsible of tracking vehicle states, by polling them every 30 seconds with a state request (HTTP GET). The core implements all the business logic:

- Process drivers requests and allocates the requested vehicle
- Manages vehicles, track their state
- Manages users (drivers signup, login)
- Manages reports and automatic tasks then it allocates them to workers

All the communications by clients (Driver, Worker, Admin) are realised in a synchronous client-server architecture, where the core is the server. The communications with the DBMS are also client-server (the core is the client) and are realised using PostgreSQL protocol. The communications with the vehicles uses HTTP but the core is the client, while vehicles are servers. The communication with the DBMS and the data modelling will be handled using an external NodeJS library, Sequelize.JS. In this way, not only we are simplifying our job, but also rely on a community tested library, i.e. more reliable.



2.2.1 Tracking vehicle state and scope

Tracking vehicle states requires a more detailed explanation. Our objective is to provide a simple interface for the core to communicate with vehicles (both tracking and controlling), so that the core can perform simple GET requests to endpoints (e.g. /vehicle/vehicle-id) to know their state (e.g. position, occupants) or POST request to perform actions on vehicles (e.g. open/close doors). Every vehicle is equipped with a wireless communication system, and is connected to the internet (so it has a public and static IP address). The core associate every vehicle ID with its IP address (and store this address in the DB), so every time a request to a specific vehicle is made, the core resolves the IP address of the requested vehicle. For security reasons, vehicles accept request only from the core IP (that is static). Every vehicle implements a simple on-board server, that answers to the core GET/POST requests, i.e. providing the vehicle's state or performing actions. Since our main objective is to design the management system, we will take as an assumption the implementation of the vehicle's on-board system. So, we could consider the Vehicle themselves as external components of the system. The team that developed the on-board system, provided us the following APIs:

- **GET /state** returns the current state of the vehicle. The state is a JSON object containing
  - Current position of the vehicle
  - Current battery charge
  - Maintenance report (if any issue is found)
- **GET /drive** returns the log of the last drive. This log includes
  - Drive time
  - Drive length
  - Start and stop positions
  - Number of occupants
- **POST /unlock** unlocks the car and start logging the drive
- **POST /lock** locks the car and stop logging
- **POST /maintenance/[on-off]** enable or disable maintenance mode, where the car disables all is security systems.

2.3 Core component

Since the other components are quite simple and atomic, only the core component need some further analysis. The core has many functionalities that are divided in different components (for component diagram, see Paragraph 2.3.1):

- **Router** This component is responsible of receiving all the HTTP request to the APIs of the core and responding to them. It routes the received request from all the clients (driver, worker, admin) and from the vehicles to the specific component, so it provides all the interfaces with the external world. For every request, the router also communicates with the Authorization Manager to check if the calling user is authorized to perform certain actions. The Router always receives HTTP data as input and parses it. All the subsequent calls to subcomponents also contains the identifier of the user that has called it. The recognize this user, the Router calls a method (checkPermission) on the AuthorizationManager.

- **Vehicle Manager** This component is responsible of tracking vehicle usage state (see state chart, paragraph 2.4.1) but also vehicle internal state (e.g. position). This states are saved to the DB to provide persistency but also analytics. It provides an interface to the Drive Controller and the Task Manager to get information about the car and to update their state. In order to be updated about the state of the vehicles the Vehicle Manager keeps polling them. The vehicle can also answer with a report of some issue, so the VehicleManager is also able to call TaskManager to generate a new report.

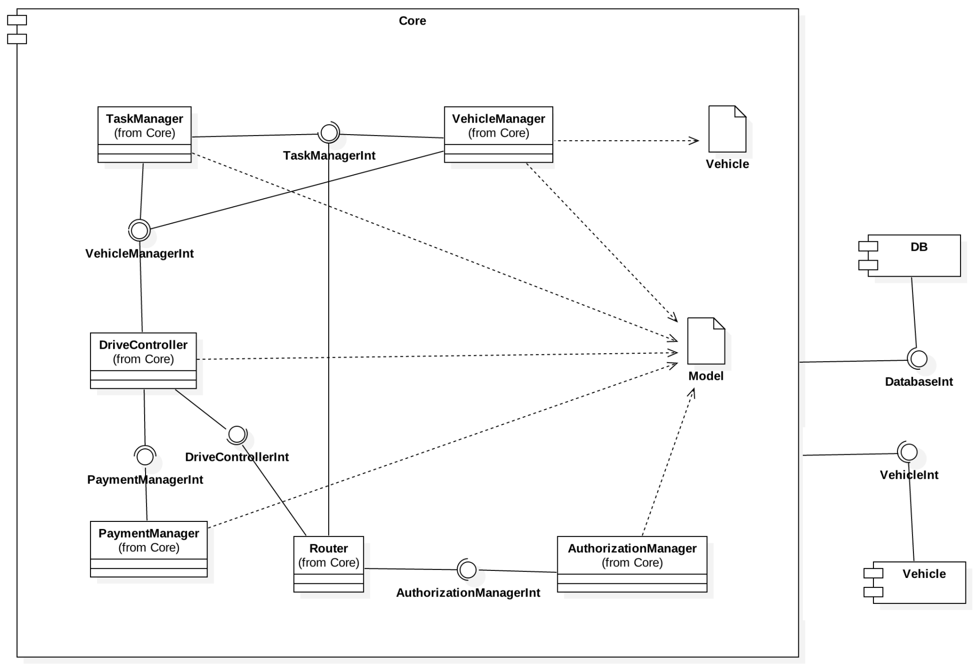
- **Drive Controller** This component is responsible of processing Drivers requests (e.g. reservation, cancellation, termination). It communicates with the Vehicle Manager to update vehicle state. It also communicates with the Payment Manager to process Drivers payments. It is also responsible of computing the correct amount due at the end of a drive. Once the drive has ended when the user has pressed the button on the vehicle, the DriveManager requests to the VehicleManager the log of the drive, it computes the base amount to pay and applies discounts. A Node.JS implementation of the amount computing is provided in chapter 3.

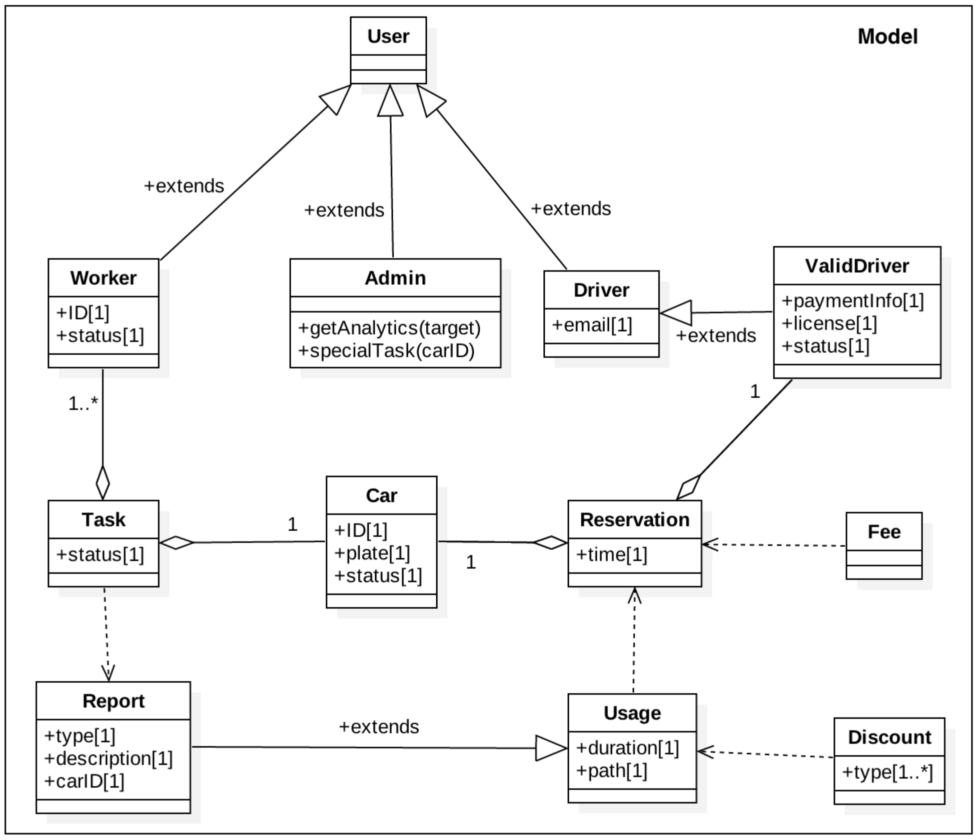
- **Payment Manager** It receives request to make a specific user pay a specific amount, process the payment and store it in the DB if the payment was successful. It is also responsible of verifying with the pre-authorization if the Driver can pay.

- **Task Manager** It is primarily responsible of managing task allocation to workers, but also of processing Drivers reports and Admins special tasks queries. It exposes an interface to the Vehicle Manager too in order to allow it to perform automatic maintenance tasks on vehicles. A Node.JS implementation of the worker choosing algorithm is provided in chapter 3.

- **Authorization Manager** It manages most of the security of the system. It is responsible of Drivers and Workers signup and login. It is also responsible of verifying that every requests to the core are performed by an authorized user (e.g. Driver can't request Workers API).

2.3.1 Core Component Diagrams

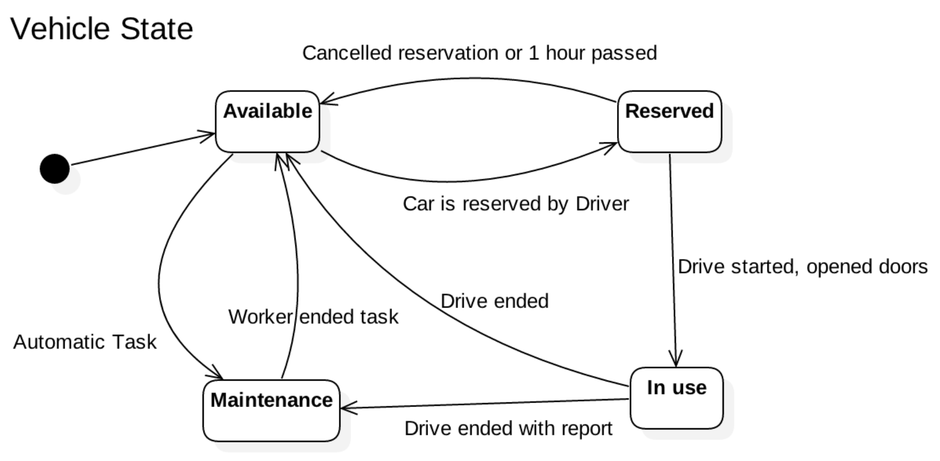




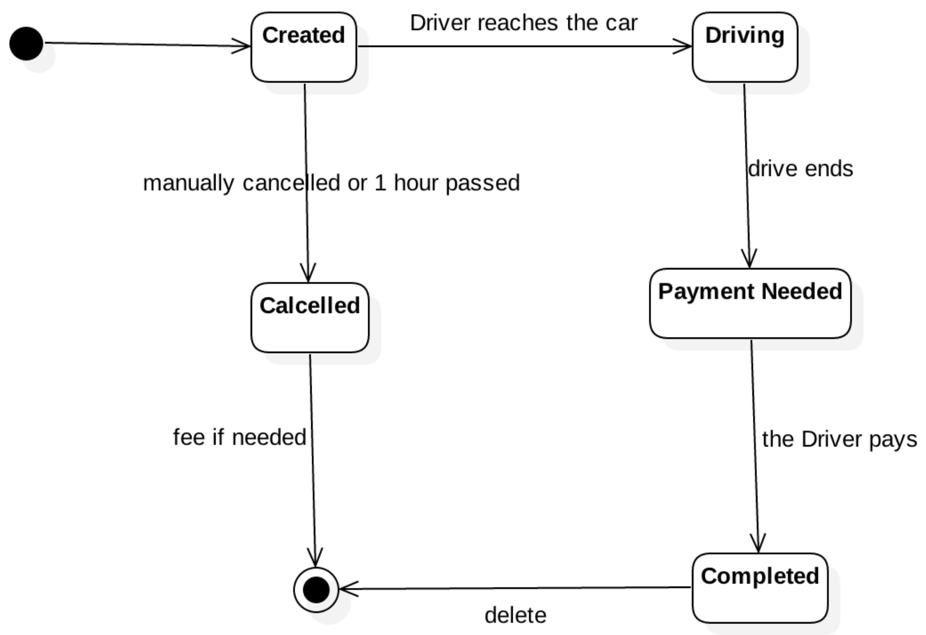
2.4 Relevant State Charts

To give a deeper explaination of the dynamic behaviour of the model we add here some State Chart Diagrams, describing the lifecycle of vehicles, reservations and tasks:

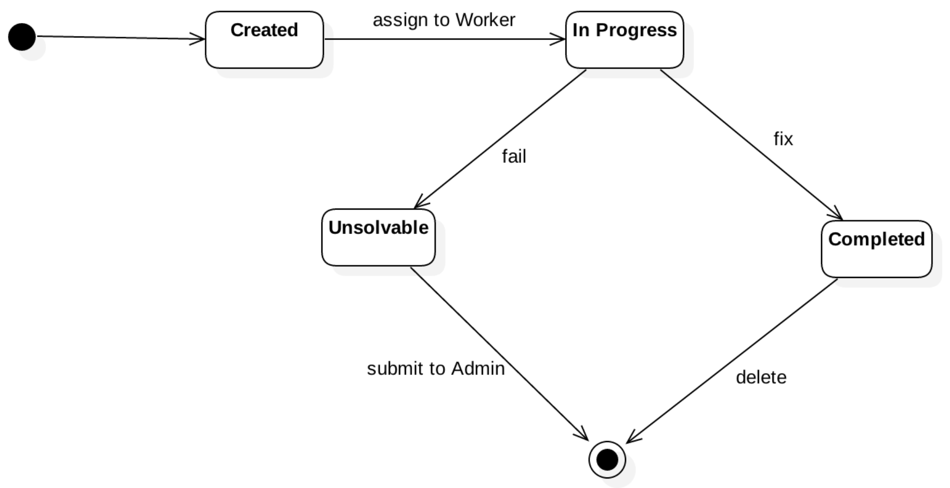
2.4.1 Vehicle State Chart



2.4.2 Reservations State Chart

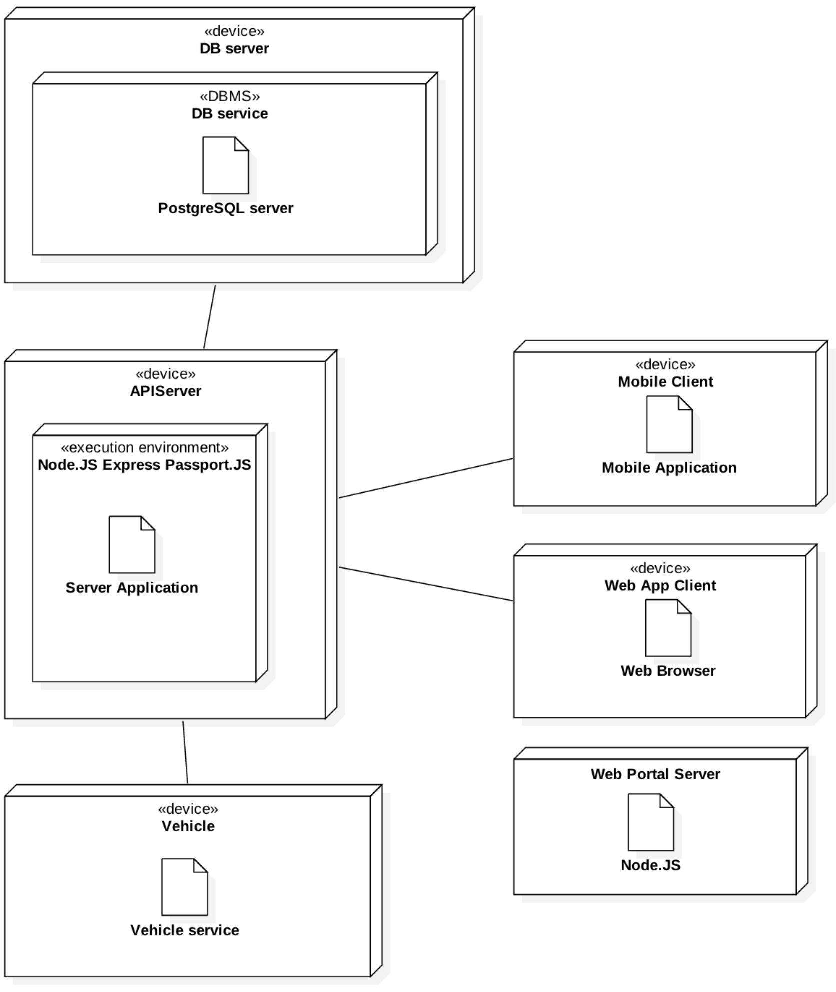


2.4.3 Task State Chart



2.5 Deployment View

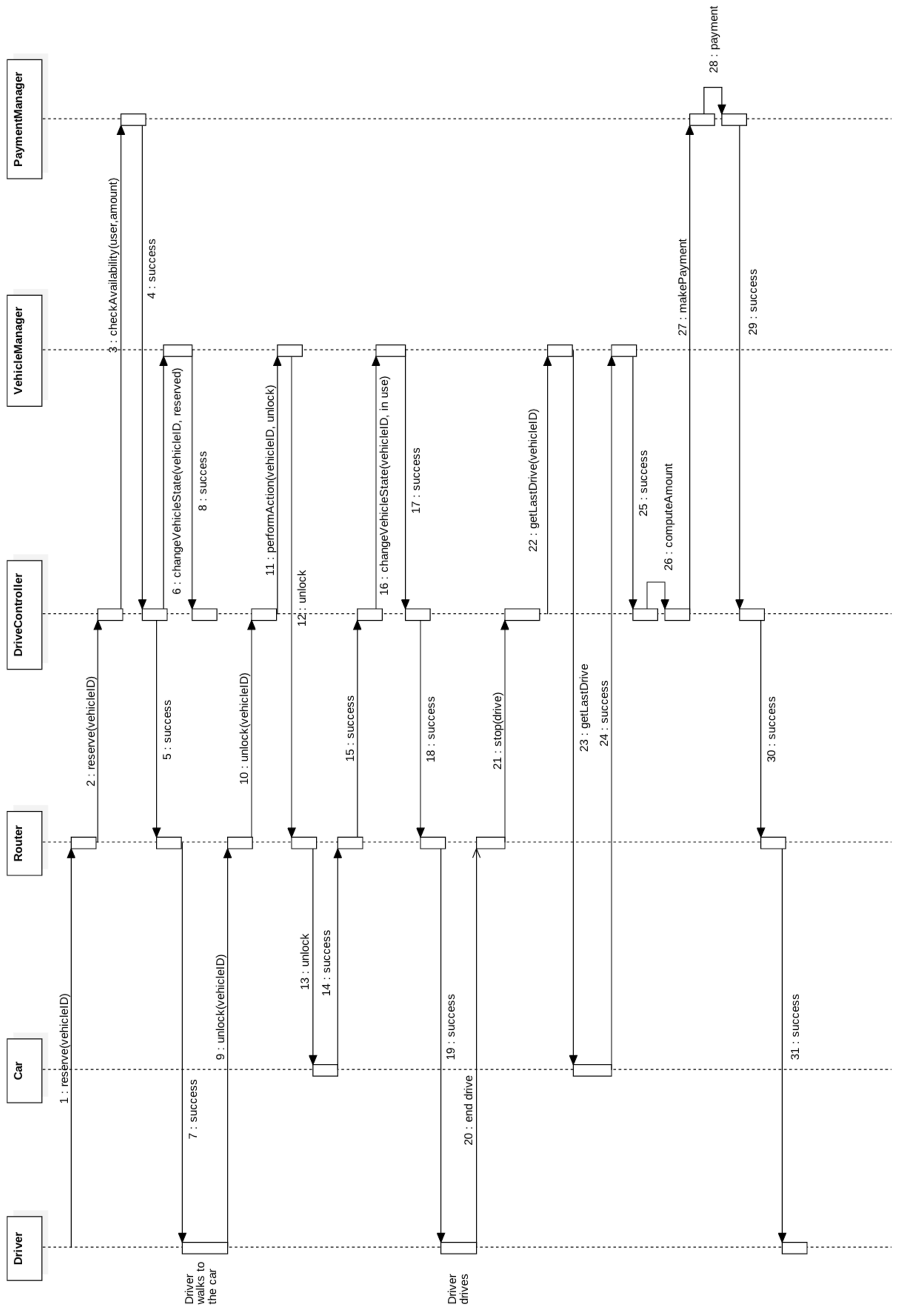
This section highlights the real deployment of the different components deepening the presentation already made in the previous paragraphs.



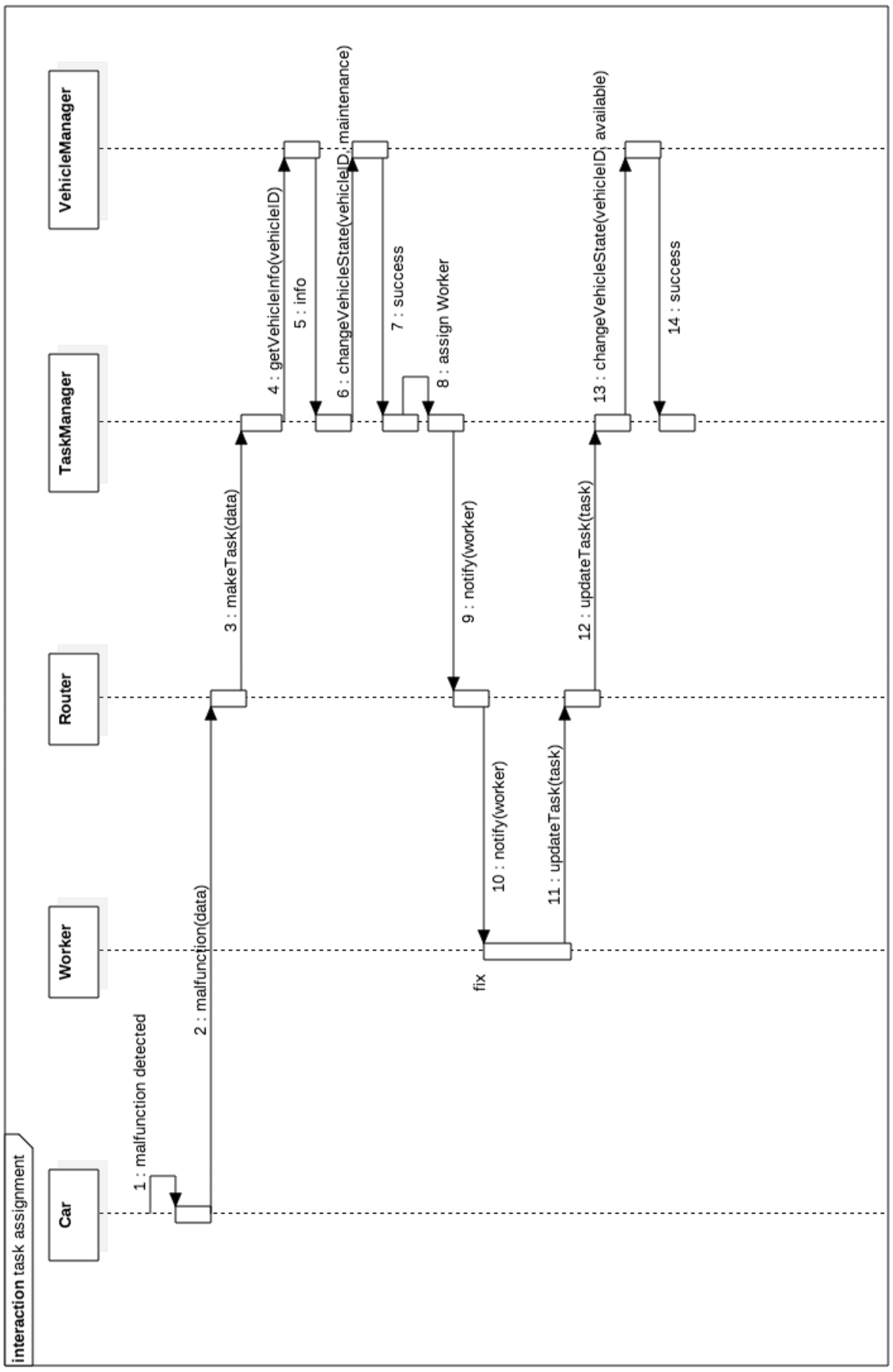
2.6 Runtime View

Below 2 UML sequence diagrams are provided, to show the dynamic behaviour during a complete drive and during a task assignment and completion.

2.6.1 Drive Sequence Diagram

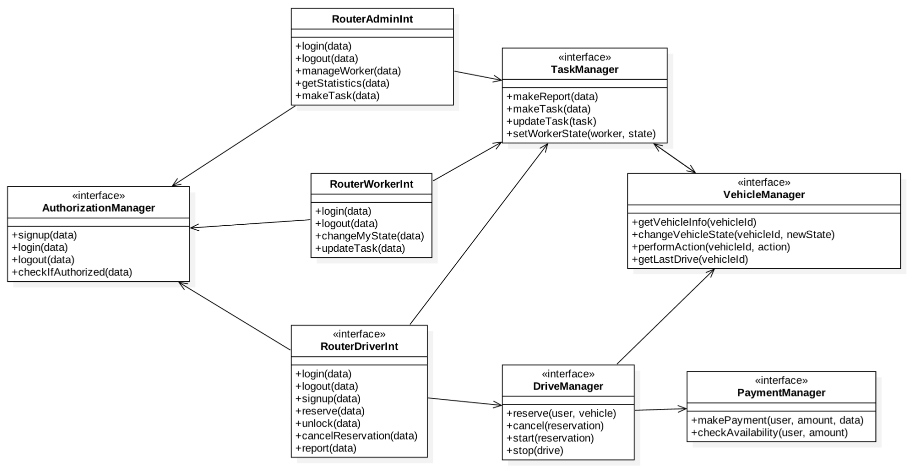


2.6.2 Task Assignment Sequence Diagram



2.7 Interfaces

Here the interface of all the subcomponents of the system are provided. The 4 interfaces of the Router are the ones exposed to clients as in the figure of section 2.3.1.



Algorithm implementation

3.1 Amount due computation

```
var computeAmountDue = function(driveData){
    //First compute the base amount
    //(depending on the duration in minutes of the drive)
    var baseAmount = data.durationInMinutes * STD_MINUTE_FARE;
    //Compute discounts as required
    var discounts = 0;
    if (data.detectedPassengers > 2){
      discounts += baseAmount * 0.1;
    }
    if (data.endBatteryLevel > 50){
      discounts += baseAmount * 0.2
    }
    if (isChargingArea(data.endPosition) &&
                data.endIsCharging){
        discounts += baseAmount * 0.3
    }
    if (nearestStationDistance(data.endPosition) > 3000
                || data.endBatteryLevel < 20){
        discounts -= baseAmount * 0.3
    }
    //Apply discount
    return baseAmount - discounts;
}
```

3.2 Task assignment

```
var assignTaskToBestWorker = function(task){
    var availableWorkers = getCurrentlyAvailableWorkers();
    distance = {};
    for (worker in availableWorker){
        //Get distance between worker and vehicle
        //with a step of 500meters
        distance[worker.id] =
                  getDist500m(task.location, worker.location);
    }
    //Select the nearest, return array of equally distant workers
    var nearestWorkers = getNearest(distance);
    //In the nearest workers
    //select the one that was inactive for more time
    inaction = {}
    for (worker in nearestWorkers) {
        inaction[worker.id] = worker.inactiveTime;
    }
    return getMax(inaction);
}
```

Requirements

4.1 Requirement Traceability

- [**GOAL1**] Driver must log in the System
  - When the Driver logs in the System using his credentials the App or the Web Portal communicate with the Router which calls the interface on the Authorization Manager to check credentials and eventually authorize the Driver.
- [**GOAL2**] Allow drivers to reserve a car up to one hour before they pick it up
  - The App provides a UI with a map to select the car to reserve;
  - Once the car is selected a request is sent to the Router which, in turn, sends it to the Drive Manager. The Drive manager calls the Payment Manager to pre-authorize the security amount and the Vehicle Manager to update the status of the car;
  - Once the Drive ends another message is sent to the Drive Manager through the Router. The Payment Manager makes the user pay and the status of the car is reset by the Vehicle Manager.
- [**GOAL3**] Allow drivers to open the reserved car
  - While the Driver reaches the car the App checks if his position is in a certain range from the reserved car. If he is nearby an unlock button is automatically shown. Clicking the button the Driver sends a request which is routed to the Drive Manager and the Vehicle Manager. Finally the Vehicle Manager sends the unlock command to the car.
- [**GOAL4**] Allow drivers to pay correctly for the service

  - When the Drive ends the car sends the recorded data to the Router which passes it to the Drive Manager to compute the correct amount. Then the Payment Manager is called to charge the Driver's credit card.

- [**GOAL5**] Allow drivers to cancel a reservation
  - The App shows a button to cancel the reservation. As always a request is sent and the reservation is deleted from the model. According to our policy, no fee is charged if the Driver cancels the reservation within an hour.

- [**GOAL50**] Worker must log in the system
  - The Worker logs in the System using his credentials;
  - The App communicates with the Router which calls the interface on the Authorization Manager to check credentials and eventually authorize the Worker.

- [**GOAL51**] Dispatch tasks to workers
  - When a Task is needed the System creates it calling the Task Manager from the Router (using RouterVehicleInt if the malfunction is detected by the car or RouterDriverInt if the report is generated by the user). When the Task is created the Task Manager assigns the most suitable Worker to the Task.

- [**GOAL52**] Allow workers to get informations about an assigned task
  - The Task Manager takes care of this, requesting car infos to the Vehicle Manager through the dedicated interface.

- [**GOAL53**] Allow workers to open and control the assigned car
  - The Authorization Manager is responsible to provide to the Worker the correct privileges.

- [**GOAL54**] Allow workers to update the state of the task (assigned, in progress, done)
  - This functionality is provided by the interface RouterWorkerInt on the Router which in turn calls the Task Manager interface to update the state of the task.

- [**GOAL100**] Log in the administration console

  - The Authorization Manager provides that functionality when called by the dedicated RouterAdminInt on the Router.

- [**GOAL101**] Allow to register new workers
  - The RouterAdminInt provides functionalities to manage Workers.
- [**GOAL102**] Allow to view the status of a specific vehicle
  - The RouterAdminInt provides functionalities to get statistics and information on Cars, Drivers and Workers.
- [**GOAL103**] Allow to request an exceptional task on a specific vehicle
  - The RouterAdminInt provides the possibility to create new tasks.

Work review

Based on our log of the work phases, the total amount of hour of work required were:

- N. Montali: 34 hours
  - Architecture
  - Algorithms
  - Interfaces
- E. Fini: 30 hours
  - Introduction
  - State diagrams
  - Sequence diagrams