Final task ISS-2021 Bologna: Automated Car-Parking

Introduction

This document deals with the design, build and development of a software system, named **ParkingManagerService**, which implements a series of functions to manage an automating parking service.

Requirements

A detailed description of the requirements is available here

Requirement analysis

Our **interaction with the customer** ha clarified that nouns have the following meanings:

- **service**: system of intercommunicating components based on a combination of hardware and software which handles interaction between human and machine;
- **automation functions**: set of functions or tasks which **ParkingManagerService** runs without any kind of employee's interaction or supervision;
- **DDR Robot**: physical or virtual (simulated in an virtual environment) object which can be represented and controlled using the infomation in basicrobot2021;
- **transport trolley**: component which uses the **DDR Robot** in order to complete his tasks;
- **home**: shelter or starting point located in the map defined in parkingMap.txt;
- **weight sensor**: virtual sensor which is able to simulate the behavior of a weight transducer which converts an input mechanical force such as load, weight, tension, compression, or pressure into another physical variable, in this case, into an electrical output signal that can be measured and converted.
  Since the system interacts with cars, the weight sensor triggers only when a sufficient weight is placed on the footplate;
- **outsonar**: physical tool capable to detect object that cross its trajectory. Data sent by the outsonar can be managed as described in SonarAlone.c;
- **parking-area**: area provided for the parking of vehicles and includes any parking spaces, ingress and egress lanes. For the purpose of the project, this area will be simulated in the **WEnv**;
- **empty room**: section of a building which is enclosed by walls and a floor. The room also does not contain anything. It isn't filled or occupied by any physical object;
- **INDOOR**: fixed point of the map which marks where car can enter the parking;
- **OUTDOOR**: fixed point of the map which marks where car can exit the parking;
- **INDOOR-area**: detailed area of the map where the weight sensor is placed;
- **OUTDOOR-area**: detailed area of the map where the out sonar is placed;
- **weight**: body(car)'s relative mass or the quantity of matter contained by it;
- **parking-slots**: fixed number (6) of place in a specific area of the map where car are or can be parked;
- **thermometer**: virtual instrument which is able to simulate the behaviour of a real sensor which is situated inside the parking and able to measuring and indicating temperature;
- **fan**: virtual object which is used to simulate the behaviour of a device for producing a current of air by the movement of a broad surface or a number of such surfaces;

- **map**: abstract representation of the environment that delineates **home**, **INDOOR**, **OUTDOOR** and **parking area**. An example of the figure is shown below:

```
|r, 0, 0, 0, 0, 0, 0, X,
|0, 0, X, X, 0, 0, 0, X,
|0, 0, X, X, 0, 0, 0, X,
|0, 0, X, X, 0, 0, 0, X,
|0, 0, 0, 0, 0, 0, 0, X,
|X, X, X, X, X, X, X, X,
```

- **fixed obstacles**: walls which marks the limit of the **room**;
- **WEnv**: virtual environment which simulate the scene of the parking in order to simplify the development in a real context. The **WEnv** configuration is reported in parkingAreaConfig.js;
- **critical situations**: anomalous situations not foreseen in advance and managed in a general way;
- **TOKENID**: set of symbols which identify in an unique way cars parked in the system and which is used in the car pick up phase;
- **ParkServiceGUI**: user interface which allows users to interact with the parking system and which shows the **SLOTNUM** and **TOKENID** and buttons which help the user to park and retrieve their car;
- **ParkServiceStatusGUI**: user interface which allows manager to interact with the parking system and which shows the **current state** of the parking area;
- **SLOTNUM**: integer that represents **parking-slots**;
- **current state**: current situation of the entire system. It includes **parking-slots** availability, room **temperature**, state of the **fan** and state of the **ParkingServiceGUI**;
- **behaviour**: status of **transport trolley**:
  - **Idle**: has no duties to perform;
  - **Working**: currently performing tasks;
  - **Stop**: state driven manually by the manager or in an automatic mechanism when **TA** > **TMAX**;
- **alarm**: signal which warn the manager of the system through the **ParkServiceStatusGUI** when the **OUTDOOR-area** has not been cleaned within the **DTFREE** interval of time;
- **proper notice**: warning sent to the user through the **ParkServiceGUI** if the **INDOOR-area** is already engaged by a car;

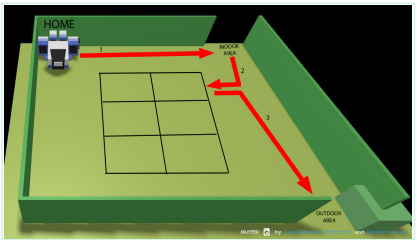
Regarding action or verbs:

- **equipped**: the system is supplied with the necessary items for a particular purpose;
- **notify/inform**: the interaction between the software system and the user and vice versa;
- **walk**: the **transport trolley** moves in order to complete its task;
- **take over**: the **transport trolley** picks up a car and carries it inside the parking;
- **observe**: the **ParkingManager** monitors the **current state** of the system;
- **clean**: the **OUTDOOR-area** is free;

Main user story

A client interacts with the **ParkServiceGUI** in order to park its car. The system, then checks if the **INDOOR-area** is free, and in that case it replies to the client with a number that identifies a free park-slot (**SLOTNUM**). At that time, if **SLOTNUM** > 0, the client receives a receipt with the **TOKENID**, enters the car and then the **ParkingManagerService** makes the transport trolley pick up the car **(1)** and moves it to the selected parking-slot **(2)**.	

When the client wants his car back, he interacts again with the **ParkServiceGUI** by requesting a pick up of his car using the **TOKENID** previously received. At this time, the transport trolley carries the selected car from its parking-slot to the outdoor-area **(3)**. Finally, the client takes the car and frees the outdoor-area.




```
Let assume the INDOOR-area is free and at least one parking slot is available:

        Client notifies interest in entering the car

        System return SLOTNUM

        assert 0 < SLOTNUM <= 6

        Client CARENTER

        System return TOKENID and park the car

        wait

        Client wants to pick up his car by sending TOKENID

        System moves the selected car to the OUTDOOR-area

        Client receives a confirm.
```

User story: normal parking manager's supervision

While the **ParkingMenagerService** is working, it collects data of system current state. This includes the temperature (**TA**), state of the fan and the **transport trolley**. The manager can view this information by the GUI.

```
System simulates the change of the current status

GUI checks if new changes appears.
```

User story: non standard parking manager's supervision

When the system is working, the thermometer measures the temperature **TA**. If **TA** > **TMAX** the **ParkingManagerService** informs the **ParkingServiceStatusGUI**. Therefore the manager requests to stop the trasport trolley and activates the fan via the GUI. The system stops the **transport trolley** and waits until **TA** < **TMAX**. When this happens the manager sees it on the GUI and requests to activate the **trasport** trolley and to stop the fan using the GUI.

```
Thermometer simulates a temperature TA > TMAX

wait

System checks new fan and transport trolley status

Thermometer simulates a temperature TA < TMAX

wait
```

```
System checks new fan and transport trolley status
```

User story: DTFREE OUTDOOR-AREA

Initially the outsonar detects the presence of a car in the **OUTDOOR-area** and refers it to the **ParkManagerService**. If the car is still waiting in the **OUTDOOR-area** within the **DTFREE** time the **ParkManagerService** sends an **ALARM** signal to the **ParkServiceStatusGui**. The manager is now aware of the situation.

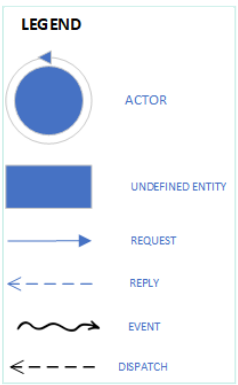
```
Outsonar simulates the presence of a car for a time T > DTFREE

ParkServiceStatusGui checks if an alarm has been received
```

Problem analysis

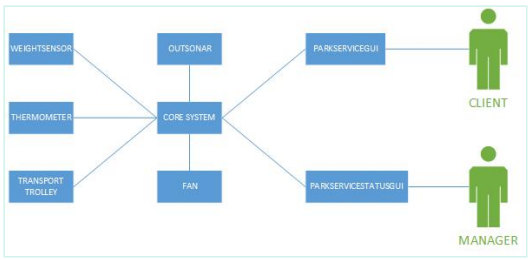
The system is made up of a series of distributed heterogeneous components that must communicate with each other. From their interaction arises the problem of evaluating the technologies to be used for the transmission of information in the best possible way. Another problem can be found in the management of the transport trolley as a core feature of the parking system.
For a first representation of the system it is possible to use the qak meta-modeling language provided by the customer, whose specifications are contained in this document.

The following legend will be used to represent the components in the problem analysis.



System overview

The side image shows a schematic view of the system, including components and users.
As can be seen from the image, the core system is the hub of the system and plays the role of mediator between components. For this reason, the central system is the same entity which elaborates information and schedules tasks for interacting components. The interaction between those entities takes place via network or via different networks using messages. Data transmission and message format will be described below.



System communication overview

Since a distributed system is being developed, communication is a major aspect of the system itself. Depending on the nature and task of each component of the system, they will need a different form of interaction. Those form of interaction can be generally expressed as **synchronous** or **asynchronous** communication.
Synchronous communication can be distinguish in:

- **request/response** semantic;
- **invitation** semantic;

Meanwhile for the asynchronous communication there are:

- **dispatch** semantic;
- **event** semantic;

Moreover, in order to use these semantic, the system can relies on different communication protocol:

- **websocket** for request/response, dispatch, invitation and event semantics;
- **HTTP** for request/response and invitation semantics;
- **MQTT** for event and dispatch semantics;
- **COaP** for request/response, dispatch, invitation and event semantics.

Afterwards, for every component will be discussed the best communication solution according to its task.

Components overview

Transport Trolley

Transport Trolley aim is reduced to two ventures:

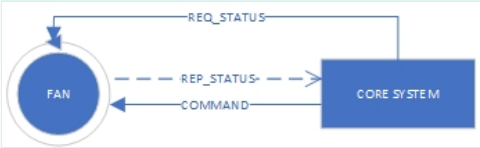
- move the client's car from the indoor-area to the parking slot assigned;
- move the client's car from the given parking slot to the outdoot-area.

At any time, parking manager is allowed to stop these tasks. Transport trolley aim leads to both active and passive behaviour.
The choice of the best path that the transport trolley has to perform is not an easy operation. For this reason, this task should be delegated to an external entity. These two entities exchange messages following a synchronous communication so that the central system receives a feedback from the trasport trolley when it asks for an execution.
Alternatively, asynchronous communication could be used. However, this type of communication would force the system to have less knowledge of the current state of the transport trolley.

Fan

The **fan** does not own or produce any valuable information by itself but it is a component controlled by the system. Fan status can be changed through two messages "**turn_on**" and "**turn_off**" respectively to turn it on and to turn it off. As a result of the command the fan can only be in two states, one is "on" which indicates that the fun is working while the second one "off" suggests that the fun is not working. The fan should also return in response the new current state in order to update the manager GUI. This feature can also be used in

case of electric disguise or general failure of the central system. The fan must respond with its current state whenever the central system requests its state of activation.



Message	Content	Semantic
req_status	status	request
command	turn_on	request
	turn_off	
rep_status	on	reply
	off	

The sensors below don't need any replies during the interaction with the core system. Also, the analysis of the requirements shows that the **outsonar**, the **thermometer** and the **weight-sensor** are autonomous active components that are unaware of the existence of other components.
This behaviour leads to an asynchronous communication that can be developed through different forms:

- **Polling**: where the core system makes, at a regular interval of time, requests to sensors;
- **Dispatches**: sent from sensor to the core system;
- **Events**: generated from sensors and sent in broadcast through networks.

There are several protocol that can be used to interact with those components:

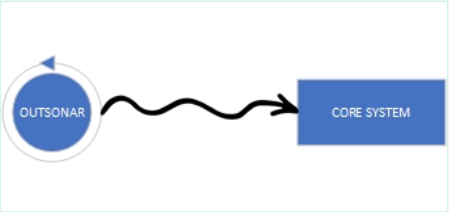
- **WebSocket**: two-way channel builded on a single TCP connection. This implies a more heavy-weight communication than the two below;
- **MQTT**: decouples producer and consumer by letting clients publish and having the broker decide where to route and copy messages. It also does best as a communications bus for live data;
- **CoAP**: a one-to-one restful protocol for transferring state information between client and server. It also meeting specialized requirements such as multicast support, very low overhead, and simplicity.

In addition, all the sensors can be modeled as actors.

Out-Sonar

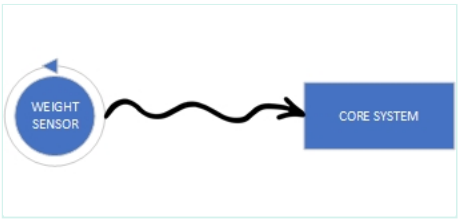
Data from the **out-sonar** is used when a client starts the **acceptOUT** procedure and to trigger the alarm procedure if it detects a car in the **OUTDOOR-AREA** for more than **DFTREE** time.
In order to identify those situation, the **out-sonar** has to emit events every time the distance measured by the sensor change. Moreover, messages sent by the **out-sonar** contain the distance misured and the relative timestamp. Customer's SonarAlone.c can be used in order to simplify the development of this component.

Message	Content	Type
event	distance, timestamp	event



Weightsensor

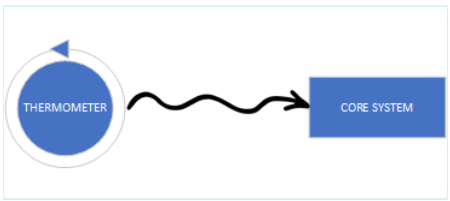
Data from **weightsensor** is used when a client starts the **acceptIN** procedure by pressing the **CARENTER** button. Like the sonar, the **weightsensor** has to emit events every time the weight measured change significantly (this allows to detect only when cars are on the sensor). In this case, events emitted by the **weightsensor** contains the weight measured and the relative timestamp in order to guarantee data freshness.



Message	Content	Type
event	weight, timestamp	event

Thermometer

Thermometer measures the temperature of the parking room and sends at regular time interval event to the core system. Since temperature doesn't change that quickly, the time interval can be quantified in the order of tens of seconds. This help the system to rely in a less bandwidth usage. The time interval must take into account of the position of the thermometer in the parking room.



Message	Content	Type
event	temperature, timestamp	event

Map

The **map** is a passive element that is used by the core system to manage the state of the parking-area. For a greater simplicity of use the map can be represented by an image composed of different symbols:

- the symbol **h** represent the "home" position of the trasport trolley;
- the symbol **0** represents an area where the trasport trolley can move;
- the numbers from **1** to **6** represent the parking slots, where the cars can be parked. The specific number is the SLOTNUM used by the core system;
- the symbol **|** represents a wall;
- he symbol **i** represents the position of coordinates (6,0), where a car is left by a client;

- the symbol **o** represents the position of coordinates (6,4), where a car is brought by the trasport trolley to be picked up by the customer.

```
|, h, 0, 0, 0, 0, 0, i, |
|, 0, 0, 1, 4, 0, 0, 0, |
|, 0, 0, 2, 5, 0, 0, 0, |
|, 0, 0, 3, 6, 0, 0, 0, |
|, 0, 0, 0, 0, 0, 0, o, |
|, |,  |, |, |, |, |,  ,|
```

GUI overview

In an attempt to achieve a wider catchment area and to guarantee simplicity and portability both GUIs should be web interfaces. In that case, **Spring** is a technology that can be used for the development of the two GUIs . Furthermore, both client and manager interfaces should be clear and user-friendly.

ParkServiceStatusGUI

Manager's interface must be able to send commands to the fan and the **trasport trolley** as well as receive and show details about current system status. Another key functionality that the manager's interface must integrate is the ability to receive and show in a proper way the alarm signal. This requirements can rely on:

- **request/response** mechanism;
- **polling** mechanism;
- **event** mechanism;

However, the polling mechanism would affect system performance and resources.
The system behaviour results in two type of interaction between the **ParkServiceStatusGUI** and the core system:

- to stop/resume both fan and trasport trolley with a request/response semantic. This allows both entities to know if a command was successful or not;
- to receive data about the current status of the system with an invitation semantic.

Below a detailed list of what the GUI must show:

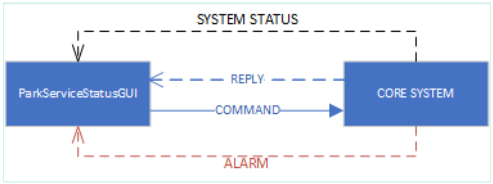
- status of the **fan** (**on/off**);
- status of the **transport trolley** (**idle/working/stopped**);
- **temperature** of the room (expressed in celsius degrees);
- status of the **INDOOR-area** (through weightsensor data);
- status of the **OUTDOOR-area** (through outsonar data);

In addiction, the GUI should contain a series of buttons to interact with:

- **ACTIVATE**/**DEACTIVATE** button for the fan;
- **STOP**/**RESUME** task button for the transport trolley;
- **STOP** button if the alarm is triggered;

The gui must also be able to capture the park manager's attention via visual and additionally audible cues when the alarm is triggered.

Message	Content	Type
Command	turn_on	request
	turn_off	



	resume_work	
	stop_work	
	stop_alarm	
Reply	ok	reply
	fail	
System Status	check above	dispatch
Alarm	alarm_situation	dispatch

ParkServiceGUI

Client's interface must be able to send commands to the core system in order to park client's car, receive a receipt and to pick up the car using the same receipt. These three phases consist in:

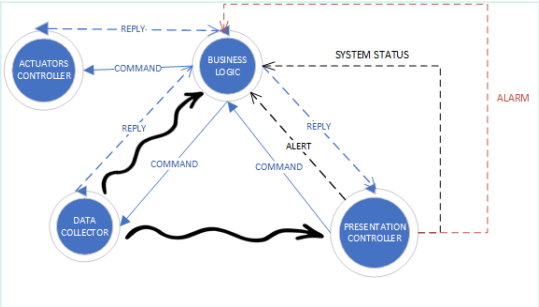
- **acceptIN**: clients manifest their will to park their car. If at least a park slot is available, the system replies with an **SLOTNUM** which identifies the park slot using an integer from 0 to 6. This interaction can take place only if the **indoor-area** is free. If **acceptIN** isn't available (**indoor-area not free) at the moment of the request, the client should receive a proper response and a future alert once acceptIN command is restored;**
- **CARENTER**: once clients leave the car in the indoor-area they press the **CARENTER** button in order to request the parking of their car and receive a **TOKENID** (an unique string of value linked to the **SLOTNUM**) as a receipt;
- **acceptOUT**: clients manifest their will to pick up their car by sending their **TOKENID** previously received to the system. If the **outdoor-area** is free and the trasport trolley is working, client requests are processed and their car moved from the parking slot to the **outdoor-area**. In addition, if the pick up isn't available once requested the system should behave like the **acceptIN** situation previously described.

Another feature that the **ParkServiceGUI** must implement is the capability of receiving an alert if the car stands at the outdoor area for more than **TMAX**.
Those mechanisms can rely on a request/response communication to send commands. Meanwhile, alerts are a type of information sent by the core system which don't require replies.

Message	Content	Type
Command	acceptIN	request
	CARENTER	
	acceptOUT	
Reply	ok	reply
	fail	
	TOKENID	
	SLOTNUM	
Alert	free_indoor_area	dispatch
	pick_up_available	
	pick_up_needed	

Core System Overview

The core system is the most important component of the entire software system because it contains the business logic. Acting like the system 'hub', it manages various types of information coming from external sensors, it receives messages or commands from clients or the manager and it controls the transport trolley and the fan. For its complexity and by following a 'divide-et-impera' approach and single responsability priciple, the core system could be divided in four different actors:



- **business logic actor**: process key functionalities for the entire system:
  - receive information from the data collector actor;
  - manage and maintain parking availability;
  - manage transport trolley's state;
  - manage automatically the fan or after a manager's command;
  - interaction between the manager and clients through the GUI controller actor and the generation and collection of the TOKENID.
- **actuators controller actor**: it interacts with components which are able to receive commands. In particular:
  - simply redirects commands from the business logic to the fan;
  - given initial and final coordinates from the business logic, generates the best path for the transport trolley to follow;
  - allows future integration with new active components much easier.
- **data collector actor**: it is responsible for receiving and elaborating information from the sensors usable by the business logic actor;
- **GUI controller actor**: it is placed between the business logic actor and GUIs. It gets information from GUIs and forwards them to the business logic actor and respectively sends datas from the business logic actor to clients' GUI.

Abstraction gap

The problem analysis was carried out to highlight the requirements without focusing on used technologies by introducing high-level concepts. Another goal was the development of a code as independent as possible from the technology used for the transport trolley in order to make it easier to replace it if necessary. The abstraction between system design and available technologies is facilitated thanks to the presence of modern programming languages such as Kotlin that allows the definition of actors. However, the use of particular technologies to deal with problems such as data persistence and failure resistance leads to a wider gap.

System weaknesses

The system may soffer some rare case studies:

- If the parking manager isn't well trained about system functionalities he may create a bottleneck of requests simply stopping transport trolley tasks through his GUI. In fact, the system isn't developed in a way that can't stop wrong decision by the parking manager;
- Considering the distributed nature of the system, network problems lead to a malfunction of the entire system;
- Some information is kept by the business logic in order to manage the system. In case of system faults, these data can be lost.

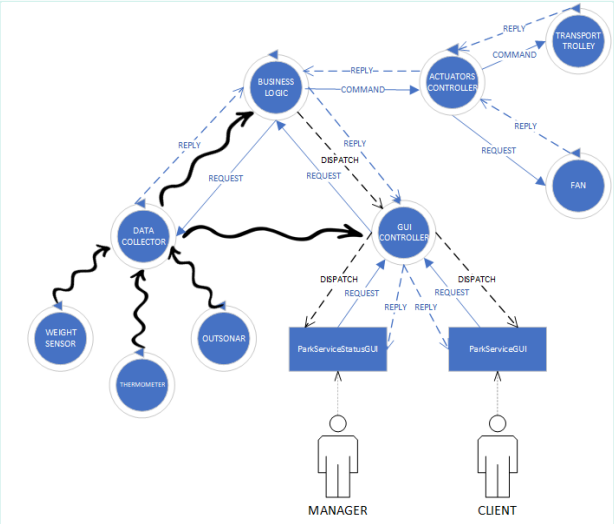
Estimate delivery time

As planned in the problem analysis all sensors and system entities including the core system can be modeled as actors. This allows the team to define a first simplified version of the system using QActors. The simplified version of the system is based on a series of assumptions:

- INDOOR cell (6,0) / OUTDOOR cell (6,4);
- temperature under **TMAX**;
- one client;
- no interaction from parking manager;
- one free parking slot (3,3) **SLOTNUM** = 6.

Following these assumptions the model can be developed in two working days. The first one will be used by a single member of the team to develop the model. The first half of the second day will be used by the team to discuss the model produced, meanwhile the second half will be used to fix and audit changes proposed previously.

Logical Architecture



The model of the system can be seen here.

In order to make our parking software reusable and as much as possibile independent from the undelying communication protocols, the designer could make reference to proper design pattern such as **adapter**,

bridge and **facade**. Other design pattern that might be considered during the development are **singleton**, **observer** and **layer**.

Test plans

In this section we discuss and bring attention to the multiple scenarios that might occur during the system viability. The object of the tests is to mock some behaviours or concrete situations and then check if the system returns what we expected.

RELEASE PHASE

During the release-phase of the car test plans must meet these solutions:

1. if at least one parking slot is available, INDOOR-AREA is free, the transport-trolley is working and the client makes a parking request, it is accepted and the client receives a number between 1 and 6 (linked to the parking slot) as response;
2. if all the parking slots are occupied the client receives number 0 as response;
3. if the INDOOR-AREA isn't available and the transport trolley is stopped the client receives a waiting message.

PICK-UP PHASE

During the pick-up phase if the TOKENID is correct, the OUTDOOR-AREA is free, the transport-trolley is working and a client requests the pick up of his car the system will provide green light as long as all these three conditions are fulfilled. If this is not the case, then a waiting message will appear.

UNCOMMON SITUATIONS

1. if the the temperature measured by the thermometer is greater than **TMAX**, the **fan** must be activate and the **transport trolley** must be stopped;
2. if a car is stuck in the **OUTDOOR-AREA** for more than **TFREE**, the system must send an alarm to the manager.

These test plans can be found here.

Work Plan

The first product backlog can be defined starting from the previous analysis. The chart contains the following items ordered by priority.

ID	PRIORITY	ITEM
1	HIGH	System processes car deposit requests
2	HIGH	System processes car pick-up requests
3	HIGH	System manages data from sensors
4	HIGH	System manages transport trolley
5	MEDIUM	System satisfies manager's start and stop fan requests
6	MEDIUM	System satisfies manager's start and stop trolley requests
7	MEDIUM	System maintains an updated version of the system status
8	MEDIUM	System recognizes critical OUTDOOR-area situation and generates an alarm for the manager
9	LOW	System asks the client to pick up his car due a DTFREE timeout

10	LOW	System manages fan automatically

All the items above can be group in several sprint as datailed in the next table:

SPRINT GOAL	BACKLOG ITEMS
System basic functionalities	• System manages data from sensors; • System manages transport trolley.
System able to manage user requests	• System processes car deposit requests; • System processes car pick-up requests.
System interaction with manager and care of system status	• System satisfies manager's start and stop fan requests; • System satisfies manager's start and stop trolley requests; • System mantains an updated version of the system status; • System recognizes critical OUTDOOR-area situation and generates an alarm for the manager.
System Optimization	• System asks the client to pick up his car due a DTFREE timeout; • System manages fan automatically.

Project

Testing

Deployment

Maintenance

By studentName email: giuseppe.cristaudo@studio.unibo.it,
filippo.manfreda@studio.unibo.it, enrico.andrini@studio.unibo.it