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

The **ParkManagerService** should create the **ParkServiceGUI** (for the client) and the **ParkServiceStatusGUI** (for the manager) and then perform the following tasks:

- **(R1) acceptIN**: accept the request of a client to park the car if there is at least one **parking-slot** available, select a free slot identified with a unique **SLOTNUM**. A request of this type can be elaborated only when the **INDOOR-area** is free, and the **transport trolley** is at **home** or working (not stopped by the manager). If the **INDOOR-area** is already engaged by a car, the request is not immediately processed (the client could simply wait or could - optionally - receive a proper notice).
- **(R2) informIN**: inform the client about the value of the **SLOTNUM**.

If **SLOTNUM** > 0:

1. **(R2A) moveToIn**: move the **transport trolley** from its current localtion to the **INDOOR**;
2. **(R2B) receipt**: send to the client a receipt including the value of the **TOKENID**;
3. **(R2C) moveToSlotIn**: move the **transport trolley** from the **INDOOR** to the selected **parking-slot**;
4. **(R2D) backToHome**if no other request is present, move the **transport trolley** to its **home** location, else **acceptIN** or **acceptOUT**.

If **SLOTNUM** == 0:

- **(R2E) moveToHome** if not already at home, move the **transport trolley** to its **home** location.

- **(R3) acceptOUT**: accept the request of a client to get out the car with **TOKENID**. A request of this type can be elaborated only when the **OUTDOOR-area** is free and the **transport trolley** is at **home** or working (**not stopped** by the manager). If the **OUTDOOR-area** is still engaged by a car, the request is not immediately processed (the client could simply wait or could - optionally - receive a proper notice).

1. **(R3A) findSlot**: deduce the number of the parking slot (**CARSLOTNUM**) from the **TOKENID**;
2. **(R3B) moveToSlotOut**: move the **transport trolley** from its current localtion to the **CARSLOTNUM**/**parking-slot**;

3. (R3C) **moveToOut**: move the **transport trolley** to the **OUTDOOR**;
4. (R3D) **moveToHome**: if no other request is present move the **transport trolley** to its **home** location;
   else **acceptIN** or **acceptOUT**

- (R4) **monitor**: update the **ParkServiceStatusGUI** with the required information about the state of the system.

- (R5) **manage**: accept the request of the manager to stop/resume the behavior of the **transport trolley**.

A detailed description of the requirements is available here

Requirement analysis

Our **interaction with the customer** has 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 parking area;
- **weight sensor**: 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. The customers gave access to SonarAlone.c which allows to manage the sonar;
- **parking-area**: area provided for the parking of vehicles and includes any parking spaces, ingress and egress lanes.
- **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**: six fixed place in a specific area of the map where car are or can be parked;
- **thermometer**: sensor which is situated inside the parking and able to measuring and indicating temperature;
- **fan**: actuator for producing a current of air by the movement of a broad surface or a number of such surfaces;
- **map**: representation of the environment that delineates **home**, **INDOOR**, **OUTDOOR** and **parking area** defined in parkingMap.txt.
- **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**: string 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;

The two actuators, following the discussion with the customer and as they are actively controlled by the ParkManagerService, can already be modeled as actors using the Qak meta-modeling language.
The model of the two actors is detailed here. In addiction, unit test for both the fan and the trasport trolley are presented.

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.

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

System communication overview

The interaction can be generally expressed as **procedure calls** or **message based**.
However, since a distributed system is being developed, communication is a major aspect of the system itself and fit best in a message based interaction.
Depending on the nature and task of each component of the system, they will need a different form of interaction.
Moreover, in order to use this semantic, the system can relies on different communication protocol:

- **websocket**;
- **HTTP**;
- **MQTT**;
- **COaP**.

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

Components overview

ParkManagerService

The ParkManagerService (**PMS**) is the core of the entire software system as it contains the business logic. Acting like the system 'hub', the **PMS** can be described as a **service**, managing various types of information coming from external sensors, receving messages or commands from clients or the manager and controlling the transport trolley and the fan.

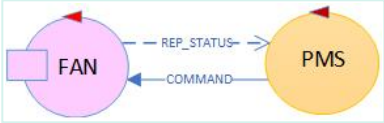
Transport trolley

Transport trolley must satisfy these requirements: R2A, R2C, R2D, R2E, R3B, R3C, R3D.
At any time, parking manager is allowed to stop these tasks. Transport trolley aim leads to both active and passive behaviour.
As discussed in the requirements analysis, the **transport trolley** is represented as an actor. This actuator receives commands by the **PMS** and replies to it with the outcome of execution. This behaviour reveals that a **synchronous communication** is more situable. 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.

Message	Content	Semantic
command	step	request
command	halt	dispatch
	forward	
	backward	
	turnLeft	
	turnRight	
reply	stepdone	reply
	stepfail	

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 by the **PMS** through two commands "**turn_on**" and "**turn_off**" respectively to turn it on and to turn it off. As a result of the command, the fan replies with its new state. This implies that 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. In case of electric disguise or general failure of the application, the fan must be able to reply with its current state whenever the **PMS** requests its state of activation. Similarly to the **transport trolley**, the interaction between the two entities is more predisposed to a **synchronous semantics**.



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

Requirements analysis shows that the **outsonar**, the **thermometer** and the **weight-sensor** are autonomous active components which emit information in a continuous way. For this reason, the sensors below don't need any replies during the interaction with the **PMS**.
This behaviour leads to an asynchronous communication that can be developed through different forms:

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

Following the discussion with the customer, each of these components isn't aware about the rest of the system. This leads to events being the best form of interaction between components and the **PMS**.

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. Moreover, as previously stated, **out-sonar** emits events which 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 which 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 events to the **PMS**. Since temperature usually doesn't change that quickly, events can be sent at regular time interval which can help the application 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 **PMS** to manage the state of the parking-area. In order to simplify the use of the map, it is 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**, **Spark** and **Grails** frameworks are technologies that can be used to simplify 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.
The system behaviour results in two type of interaction between the **ParkServiceStatusGUI** and the **PMS**:

- 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. Alternatively, a dispatch semantic can be used to make the interaction less constrained;
- to receive data about the current status of the system with a dispatch or invitation semantic.

Below a detailed list of what the GUI must show:

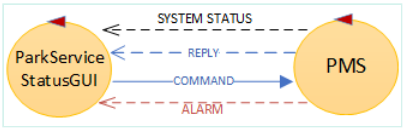
- status of the **fan** (**on**/**off**);

- 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 **PMS** 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 R1, R2B and R3 requirements. These requirements are satisfied using a request/response communication. This type of communication is needed due to the fact that the client needs a feedback everytime he asks for a service.
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**. Alerts are a type of information sent by the **PMS** which don't require replies.

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

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.

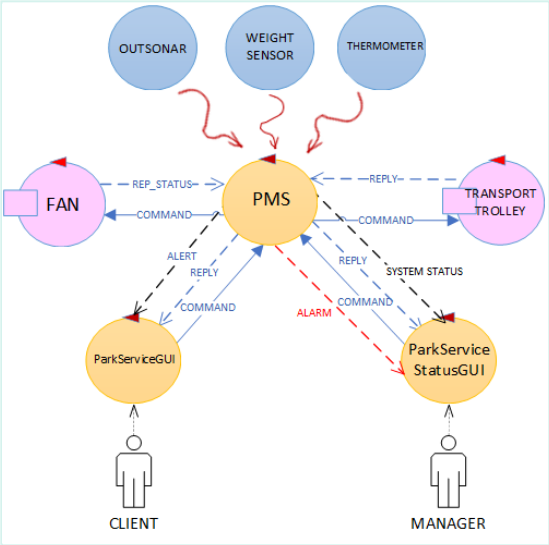
System weaknesses

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.

Logical Architecture

From the previous considerations on the components of the system it is possible to define a first logical architecture:



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**.

From the different user stories given by the customer, a model of the first phase (**client - parking phase**) will be developed using the Qak meta-modelling language. 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;
- free INDOOR-area;
- no interaction from parking manager;
- one free parking slot (3,3) **SLOTNUM** = 6.

Estimate delivery time

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.

The model is available here.

Test plans

Features presented in **client - parking phase** user story are tested following four test plans. These tests have been defined according with assumptions made before and some different configuration:

1. Slot 6 free. Client makes a request to enter, it is accepted by the **PMS** and the client receives **6** (linked to the parking slot) as response;
2. all the parking slots are occupied . Client makes a request to enter and receives number **0** as response;
3. the INDOOR-AREA isn't available. Client makes a request to enter receives a waiting message.
4. INDOOR-AREA free. Client makes **CARENTER** request and receives **TOKENID** as response.

Test plans are based on the model developed and are available 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	REQUIREMNTS
1	HIGH	**PMS** processes car deposit requests	R1, R2, R2B
2	HIGH	**PMS** processes car pick-up requests	R3
3	HIGH	**PMS** manages data from sensors	R1, R2, R3
4	HIGH	**PMS** manages transport trolley	R2A, R2C, R2D, R2E R3A, R3B, R3C, R3C
5	MEDIUM	**PMS** satisfies manager's start and stop fan requests	R4
6	MEDIUM	**PMS** satisfies manager's start and stop trolley requests	R5
7	MEDIUM	**PMS** maintains an updated version of the system status	R4
8	MEDIUM	**PMS** recognizes critical OUTDOOR-area situation and generates an alarm for the	R4

		manager	
9	LOW	**PMS** asks the client to pick up his car due a DTFREE timeout	
10	LOW	**PMS** manages fan automatically	

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

SPRINT GOAL	BACKLOG ITEM IDS
PMS basic functionalities	**3**, **4**
PMS able to manage user requests	**1**, **2**
PMS interaction with manager and care of system status	**5**, **6**, **7**, **8**
PMS Optimization	**9**, **10**

Project

Testing

Deployment

Maintenance

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