Final task ISS-2021 Bologna: Automated Car-Parking

Requirements

Automated Car-Parking

A company intends to build an *automating parking service* composed of a set of elements:

- A software system, named *ParkManagerService*, that implements the required automation functions.
- A **DDR** robot working as a *transport trolley*, that is intially situated in its *home* location. The **transport trolley** has the form of a square of side length *RD*.
- A *parking-area* is an empty room that includes;
  - an *INDOOR* to enter the car in the area. Facing the **INDOOR**, there is a *INDOOR-area* equipped with a *weigthsensor* that measures the *weigth* of the car;
  - an *OUTDOOR* to exit from the **parking-area**. Just after the **OUTDOOR**, there is *OUTDOOR-area* equipped with a *outsonar*, used to detect the presence of a car. The **OUTDOOR-area**, once engaded by a car, should be freed within a prefixed interval of time *DTFREE*;
  - a number **N (N=6)** of *parking-slots*;
  - a *thermometer* that measures the temperature *TA* of the area;
  - a *fan* that should be activated when **TA > TMAX**, where *TMAX* is a prefixed value (e.g. **35**)

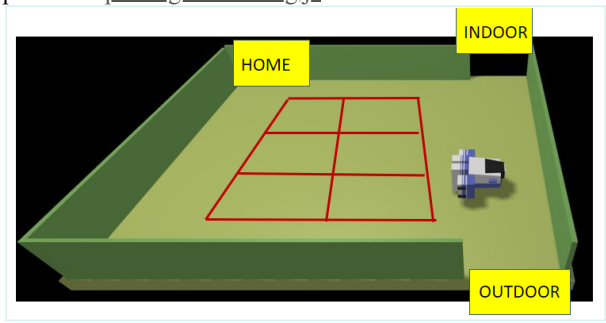
  A *map* of the parking area, represented as a grid of squares of side length *RD*, is available in the file parkingMap.txt:

```
|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,
```

  The map includes the positions of the **parking-slots** (marked above with the symbol **X**) and of the *fixed obstacles* in the area (the walls marked with the symbol **X**).

  The area marked with **X** is a sort of 'equipped area' upon which the **transport trolley** cannot walk. Thus, to get the car in the **parking-slot (2,2)**, the **transport trolley** must go in cell **(1,2)**.

  The proper scene for the WEnv is reported in: parkingAreaConfig.js



- a *parking-manager* (an human being) which supervises the state of the **parking-area** and handles critical situations.

> The job of our company is to design, build and deploy the *ParkManagerService*.

User stories

As a **client - parking phase** :

- I intend to use a *ParkServiceGUI* provided by the **ParkManagerService** to notify my interest in *entering* my auto in the **parking-area** and to receive as answer the number *SLOTNUM* of a free parking-slot (**1<=SLOTNUM<=6**). **SLOTNUM==0** means that no free slot is available.
- If **SLOTNUM >0**, I move my car in front to the **INDOOR**, get out of the car and afterwards press a *CARENTER* button on the *ParkServiceGUI*. Afterwards, the **transport trolley** takes over my car and moves it from the **INDOOR** to the selected **parking-slot**. The **ParkServiceGUI** will show to me a receipt that includes a (unique) *TOKENID*, to be used in the *car pick up* phase.

As a **client - car pick up phase** :

- I intend to use the **ParkServiceGUI** to submit the request to pick up my car, by sending the **TOKENID** previously received.
- Afterwards, the **transport trolley** takes over my car and moves it from its **parking-slot** to the **OUTDOOR-area**.
- I move the car, so to free the **OUTDOOR-area**.

As a **parking-manager**:

- I intend to use the *ParkServiceStatusGUI* provided by the **ParkManagerService** to observe the *current state* of the **parking area**, including the value **TA** of the temperature, the state of the **fan** and the state of the **transport trolley** (**idle, working or stopped**).
- I intend to *stop* the **transport trolley** when **TA > TMAX**, activate the **fan** and wait until **TA < TMAX**. At this time, I stop the **fan** and resume the behavior of the **transport trolley**. Hopefully, the **start/stop of the fan** could also be automated by the **ParkManagerService**, while the **start/stop of the transport trolley** is always up to me.
- I expect that the **ParkManagerService** sends to me an *alarm* if it detectes that the **OUTDOOR-area** has not been cleaned within the **DTFREE** interval of time.

Requirements

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

- *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).
- *informIN*: inform the client about the value of the **SLOTNUM**.

If **SLOTNUM>0**:

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

If **SLOTNUM==0**:

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

- *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. *findSlot*: deduce the number of the parking slot (*CARSLOTNUM*) from the **TOKENID**;
  2. *moveToSlotOut*: move the **transport trolley** from its current localtion to the **CARSLOTNUM/parking-slot** ;
  3. *moveToOut*: move the **transport trolley** to the **OUTDOOR** ;
  4. *moveToHome*: if no other request is present move the **transport trolley** to its **home** location;
     else *acceptIN* or *acceptOUT*

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

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

About the devices

All the sensors (**weigthsensor**, **outsonar**, **thermometer**) and the **fan** should be properly simulated by mock-objects or mock-actors.

When available a Raspberry and a sonar

The **outsonar** could be a real device. We can simulate the presence/absence of a car.

Non functional requirements

1. The ideal work team is composed of **3 persons**. Teams of 1 or 2 persons (**NOT** 4 or more) are also allowed.
2. The team must present a **workplan** as the result of the requirement/problem analysis, including some significant **TestPlan**.
3. The team must present the sequence of **SPRINT** performed, with appropriate motivations.
4. Each **SPRINT** must be associated with its own 'chronicle' (see templateToFill.html) that presents, in concise way, the key-points related to each phases of development.
   Hopefully, the team could also deploy the system using docker.
5. Each team must publish and maintain a GIT-repository (referred in the templateToFill.html) with the code and the related documents.
6. The team must present (in synthetic, schematic way) the **specific activity of each team-component**.

Requirements analysis

Glossary

- **Transport trolley:**
  - a squared robot of side **RD** able to pick up a car and transport it from a point of the map to another;
  - the robot picks up and puts down cars from the square in front of it;
  - while the car is on the robot, they fill the same space on the map;
- **Home:** the home location of the **trolley** is the north-western corner of the parking-area, facing south;
- **Parking-area:** rectangular empty room containing the **parking-slots**, the home location and some maneuvering space for the **trolley**;
- **INDOOR-area:** the area where cars wait for the **trolley** to pick them up; it is not part of the **parking-area**;
- **OUTDOOR-area:** the area where the **trolley** leaves the cars to be picked up by their owners; it is not part of the **parking-area**;
- **Parking-slot:** a squared portion of the **parking-area** of side **RD** where parked cars are stored by the **trolley**; there are six of them and they are identified by a **SLOTNUM**;
- **Fan:** device able to lower the temperature of the **parking-area**;
- **Fixed obstacles:** the parts of the **parking-area** where the **trolley** is not able to pass through;
- **Hopefully / Optionally:** the corresponding requirements should be implemented and employable as an alternative to the default behavior.

Provided software and hardware

The transport trolley

The **trolley** is a direct-drive robot compatible in terms of interaction and behavior with the virtual environment WEnv provided by the customer.

Since every position in the **parking-area** is expressed as a multiple of the side **RD** of the **trolley**, the communication with the virtual robot should follow the aril convention.

The parking-area and its points of interest

The customer provided a **map** of the **parking-area** in a **textual format**. Said map is partially compatible with an available legacy tool called PlannerUtil, which is able to plan the route to drive a robot to a specified spot avoiding obstacles. Said tool, however, requires the **map** to be in a **binary format**. It should be made available as soon as possible a utility function to convert the representation of the **map**.

The customer provided also a second description of the parking-area as a **JavaScript** configuration file compatible with the **WEnv**. An installation of the **WEnv** with said configuration should be made available as soon as possible for testing purposes.

All the sensible spots in the **parking-area** are representable as **pairs** of coordinates on the map. However, while being not walkable over, the vast majority of them require a specific position and orientation of the **trolley**. It is indeed convenient to represent them not by their position, but by a **triple** representing the required **position** (in terms of X and Y) and **orientation** (in terms of N, E, S or W) of the **trolley** to properly operate on them.

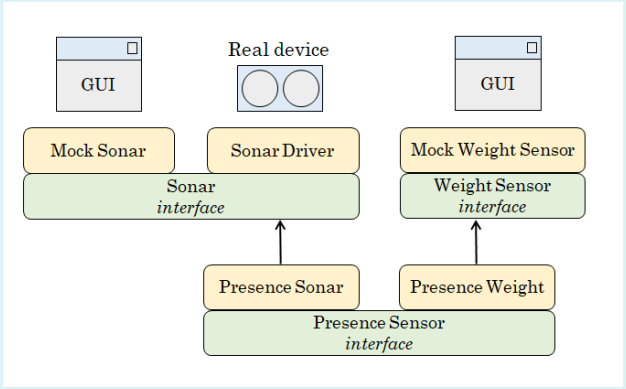
```
Home Location:  [0, 0, S]
INDOOR-area:    [6, 0, N]
OUTDOOR-area:   [6, 4, S]
Parking-slot 1: [1, 1, E]
Parking-slot 2: [1, 2, E]
Parking-slot 3: [1, 3, E]
Parking-slot 4: [4, 1, W]
Parking-slot 5: [4, 2, W]
Parking-slot 6: [4, 3, W]
```

The **PlannerUtil** should be extended by a new component to add the functionality to plan the desired orientation of the trolley alongside its desired position.

Weightsensor and outsonar

The customer clarified that the **weightsensor** measures the **weight of the car** in the **INDOOR-area** and the **outsonar** detects the **presence of a car** in the **OUTDOOR-area**, but he doesn't provide code or physical devices to use. However, he specified that both should be properly simulated by mock-objects or mock-actors.

While being two physically different components, the **weightsensor** and the **outsonar** serve the same logical function and thus they can be both modeled as **presence sensors** with appropriate **adapters**. We report hereunder a diagram to illustrate the conceptual taxonomy:



These sensors should be **queryable entities** as to know if they does or does not detect a presence. They should also be put behind some kind of interface.

Concerning the additional requirement to use a real device for the **outsonar**, the customer clarified that he expects it to be a **HC-SR04** distance sensor mounted on a **Raspberry Pi** single-board computer. We have some legacy control software available for this use case.

Thermometer and Fan

The customer clarified that the **thermometer** measures the **temperature of the area** and about the **fan** he doesn't specify its behavior. Furthermore, he doesn't provide code or physical devices to use. However, he specified that both should be properly simulated by mock-objects or mock-actors.

Much like the sonar and the weight sensor, the **thermometer** should be a **queryable component** put behind some kind of interface. The **fan** should be a passive component able to start and stop the device when **notified** to do so and it should be put behind an interface, too.

ParkServiceGUI and ParkServiceStatusGUI

The **ParkServiceGUI** (used by clients) and the **ParkServiceStatusGUI** (used by the parking-manager) must be provided by our **ParkManagerService**.

The customer has intended that the two GUIs must be very available entities able to function on many devices. They are likely best to be designed as **Web GUIs**.

User stories

The user stories provided by the customer are sufficiently precise and complete and they do not need immediate elaboration.

Informal test plan

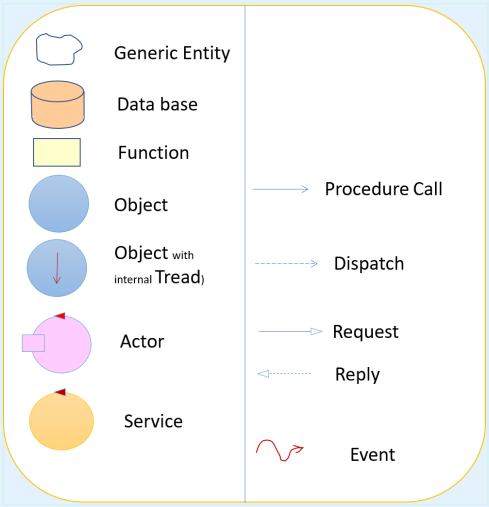
The requirement details provided by the customer are accurate enough to act as an informal test plan on their own.

Problem analysis

Relevant aspects

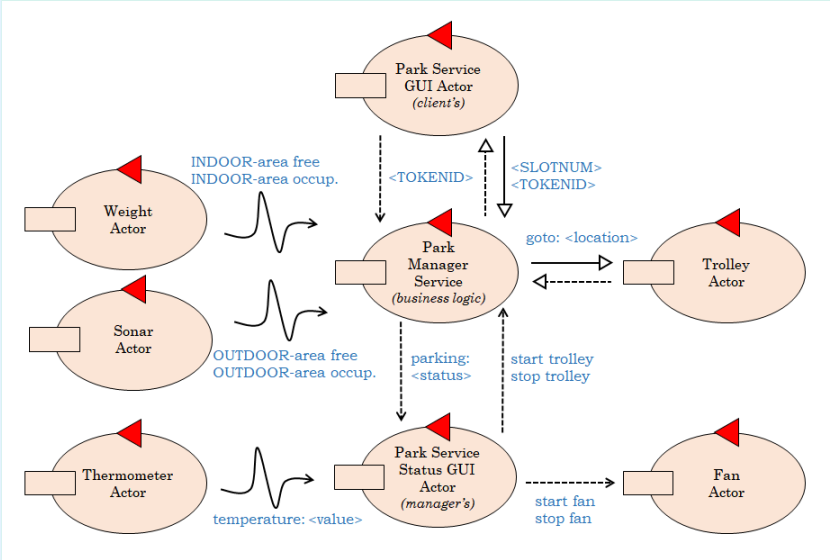
1. The system to be built will be a distributed system, consisting of several macro-components:
   - the **transport trolley**;
   - the **weight sensor**;
   - the **thermometer**;
   - the **fan**;
   - the **outsonar**;
   - our **application** (ParkManagerService, complete with its two GUIs), which sends commands to the trolley in order to meet the requirements.
2. This distributed system, made of several heterogeneous components, would benefit greatly from an **actor-based** framework with support for **message-passing** interaction; for this purpose, we should leverage the available QAK meta-model to build executable models;
3. There isn't any conceptual abstraction gap for this problem, however since we proposed to exploit the **QAK meta-model**, we are put in front of an **abstraction gap** regarding the use of languages because QAK relies on Java and Kotlin to work. However, QAK itself manages to fill a large part of this limitation, as it is supplied with its own domain specific language and because it was designed specifically for heterogeneous distributed systems;

4. The **QAK meta-model** provides compatibility with the communication protocols **TCP**, **MQTT** and **CoAP**, equally valid for the interaction between actors; regarding the communication with the **WEnv** (or other compatible trolleys), both supported interaction models (**HTTP POST** and **WebSocket**) are fit for the task.
5. We will use the following **legend** for all the diagrams in the document:

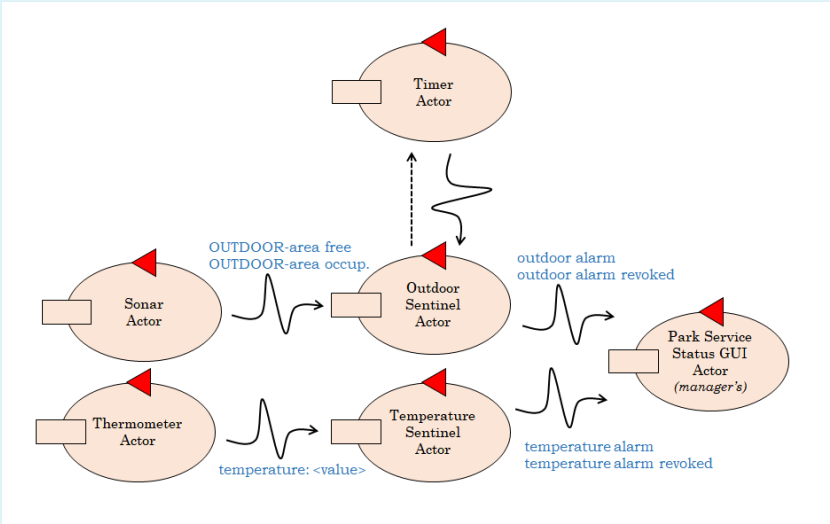


Logical architecture

Each logical component should be modeled as an actor or split in two or more actors as expressed by the following general architecture:

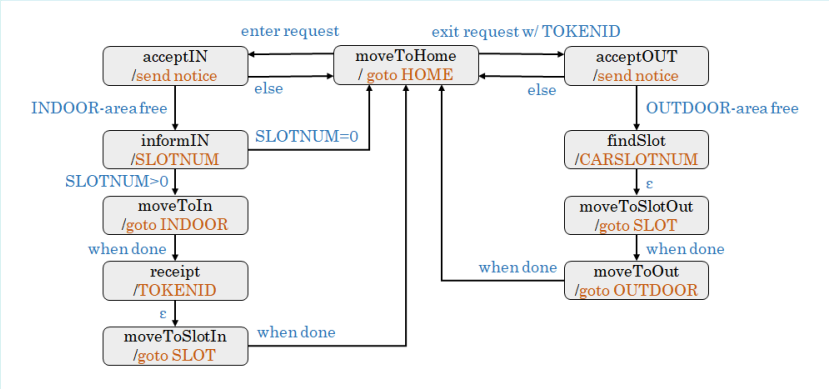


Moreover, here's the conceptual architecture of the alarms sub-system:



The business logic

The core business logic can be conveniently described as a finite-state machine to be enclosed into a proper actor. Such actor controls the main state of the application and presents a purely **reactive** behavior, since it just reacts to external stimuli.



The **goto** outputs should be delivered to an entity able to perform the implied set of elementary actions on the **trolley**.

The additional task **manage** must ensure that, regardless of the current state, the execution of the automaton gets **stopped** whenever the manager gives the relative signal. When the manager gives the resume signal, the automaton must resume its evolution from where it was interrupted. Moreover, the additional task **monitor** must update the **ParkServiceStatusGUI** with the current status of the system.

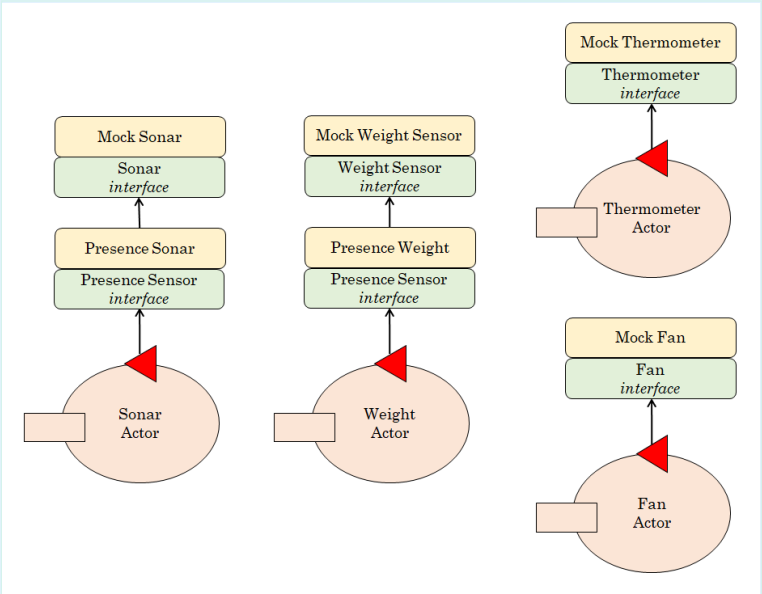
In states **acceptIN** and **acceptOut**, if the conditions to process the relative requests are not met, their processing must be postponed. While the **trolley** is stopped, new enter and exit requests by clients are not postponed, they are instead refused (with proper notice).

Sensors and actuators

Each actor using a **presence sensor** entity should poll the presence within a fixed interval of time and fire an **event** each time the status changes from present to not-present (or the other way around). These actors thus present **proactive** behavior.

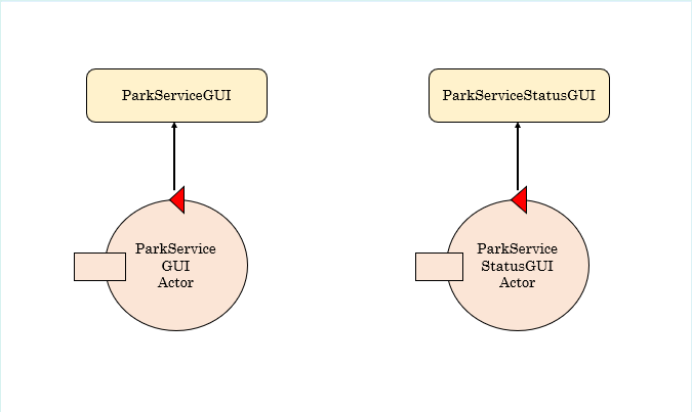
Much like the presence sensors, the **thermometer** entity should be encapsulated into a **proactive** actor that polls the temperature and fires an **event** with the new temperature value each time it varies **significantly** according to the sensitivity of the device.

The **fan** actor is **reactive** in nature and should be able to receive **dispatches** and update the status of the fan entity according to them.



The user interfaces

The **ParkManagerService** app is composed of two GUIs: the **ParkServiceGUI**, used by clients, and the **ParkServiceStatusGUI**, used by the parking-manager. The GUIs should be controlled by two different and independent actors because they may not be hosted on the business logic node. They will probably not even run on the same node, because the physical device used by the client will likely not be the same device used by the manager. If the designer chooses to build them with a Web framework, then the GUIs will act as websites and this constraint will probably not be required.

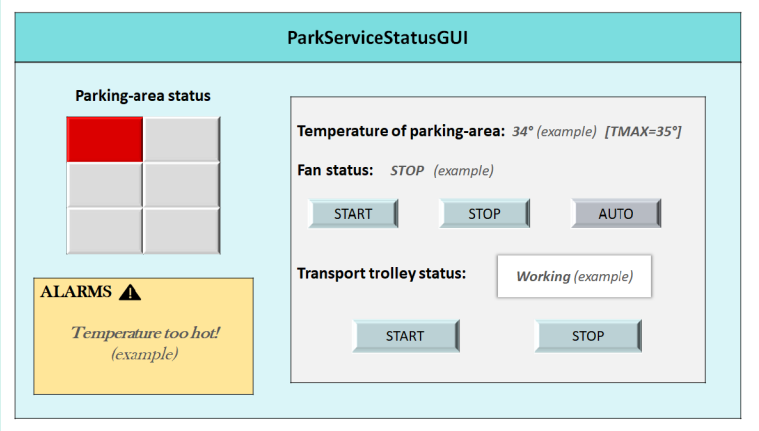


The **proactive** behavior (toward the system) is prevalent on both GUIs, since their main purpose is to take in user input. Especially the **ParkServiceStatusGUI** presents however also a **reactive** behavior in how it handles alarms.

The **ParkServiceGUI** has three buttons:

1. The **PARKING SLOT** button is needed to notify the interest by a client in **entering** its car in the **parking-area** and to receive the number **SLOTNUM** of a free **parking-slot** (between 1 and 6). If **SLOTNUM** is equal to 0, it means that no free slot is available.
2. The **CARENTER** button can be pressed only if **SLOTNUM** is greater than 0, so that the **transport trolley** may take the car from the **INDOOR** to the assigned **parking-slot**. Then it would show the client a receipt that includes the unique **TOKENID** that will be needed later to pick up the car.
3. The **PICK UP THE CAR** button is needed to **pick up** the car with the **TOKENID** received before, which needs to be written in its specific section under the button. The client's car will then be moved to the **OUTDOOR-area**.

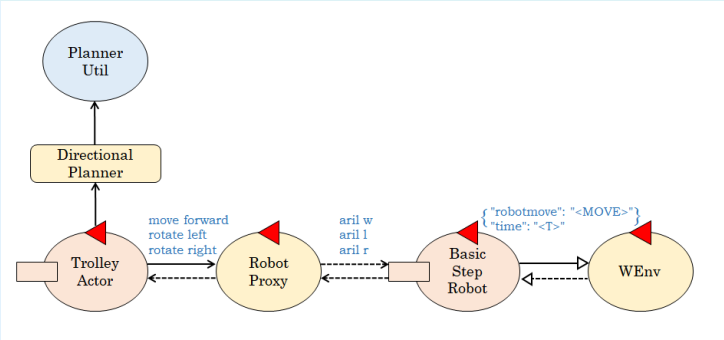
The **ParkServiceStatusGUI** has a grid with the **parking-area** status (red rectangles are occupied **parking-slots**), a section with the **transport trolley** status (idle, working or stopped) and a dedicated section for the **temperature**, in which you can see and manage the **fan** status when it is necessary with the use of three buttons: **start**, **stop** and **auto** (to automate the behavior of the fan); here you can also suspend and resume the behavior of the **trolley** with the **start** and **stop** buttons. At last, there is also a rectangle in which **alarms** appear.



Driving the trolley

To access the **WEnv**, it should be considered the usage of a more sophisticated tool, already available, called BasicStepRobotActor, which would be able to avoid interferences among overlapping commands and to natively receive movement orders expressed as steps. A new service, for now called **RobotProxy**, may be required to ensure compatibility between our system and the legacy tool **BasicStepRobotActor**.

The diagram hereunder presents a possible architecture taking advantage of the suggested tools **BasicStepRobotActor** and **DirectionalPlanner** (extension of **PlannerUtil**):



The **WEnv** requires a **request-reply** communication paradigm. Even if some intermediate steps will probably favor different approaches, the replies from the **WEnv** should nevertheless be always brought back to the actor that originated the relative requests.

The **WEnv** itself presents both proactive and reactive behavior, however it is only required its **reactive** nature for this application, since there are no virtual sonars in the scene and since robot collisions are theoretically excluded thanks to the map.

The alarms

The system requires the presence of a timed alarm for when the **OUTDOOR-area** has not been cleared within **DTFREE** time from the beginning of its occupation. The system also suggests the presence of a second alarm for when **TA** exceeds **TMAX** and some kind of notification for when **TA** goes back into its normal low temperature range.

We suggest the presence of two **proactive** actors, one for the **OUTDOOR** and one for the temperature, that fire an **event** each time they have an alarm or a notification to report. For the actor requiring temporization, it is available a legacy component called TimerActor that may be leveraged for this purpose (see the alarms sub-system diagram).

Refined test plan

User story 1 - parking phase

- In case of occupied INDOOR-area, when the client presses the **PARKING SLOT** button he receives a notification that no request can be sent.
- The INDOOR-area is free and the client uses the **PARKING SLOT** button, the business logic processes the request and the SLOTNUM is displayed in the GUI itself.
- If SLOTNUM > 0, then the client presses the **CARENTER** button and the transport trolley positions itself in the INDOOR-area and then in the slot previously indicated by SLOTNUM. Then that specific parking slot changes status and is shown as occupied in the ParkServiceStatusGUI table. Finally, the transport trolley returns to its home. At that point, a receipt with a TOKENID is shown to the client on the screen.
- If SLOTNUM = 0, a message is shown on the screen indicating that there is no space in the parking-area and, if the client decides to wait for its turn, as soon as SLOTNUM > 0 then his request is processed as in the previous case.
- If the transport trolley status is stopped and the client presses the **CARENTER** button, the request is rejected and the client is notified.

User story 2 - car pick up phase

- When the client presses the **PICK UP THE CAR** button, the request to enter the TOKENID is shown on the screen. If the code entered exists and is correct and the OUTDOOR-area is free, the transport trolley goes to the parking-slot (associated to that TOKENID) and then to the OUTDOOR-area. Finally it returns to home. Then that specific parking-slot will change status and will be shown as free in the ParkServiceStatusGUI table.
- If the OUTDOOR-area is occupied, when the client presses the **PICK UP THE CAR** button, he receives a notification that no request can be sent. As soon as the area is free, his request will be processed as in the previous case.
- If the entered code does not exist or is not correct, then an error message will be shown on the screen and the client will have to re-enter the TOKENID.
- If the transport trolley status is stopped and the client presses the **PICK UP THE CAR** button, the request is rejected and the client is notified.

User story 3 - parking-manager

- When the parking manager presses the **START** button in the fan area, the mock of the fan declares to have turned on and the fan status changes to START. When the **STOP** button is pressed and the fan declares that it has turned off, the fan status

changes to STOP.

- When the **AUTO** button has been pressed and the temperature rises above TMAX, the fan activates itself and the fan status changes to START. When the temperature value returns below TMAX, the fan turns off and the fan status changes to STOP.
- When the parking manager presses the **START** button in the transport trolley area, the latter stops working and its status in the ParkServiceStatusGUI changes to START. When he then presses the **STOP** button, the transport trolley resumes its work and its status becomes STOP.
- When the OUTDOOR-area remains occupied more than the DTFREE time interval, then an alarm message is shown on the screen. As soon as the OUTDOOR-area is free again, the error message is no longer shown.
- When the temperature is higher than TMAX, an alarm message is shown on the screen. As soon as the temperature is again lower than the threshold value, the error message is no longer shown.

Executable model

A QAK executable model of the logical architecture can be retrieved from the following local link: *trolley.qak* (copy on GitHub here). Every actor introduced in the main logical architecture and in the alarms sub-system has been implemented.

The QActor **parkmanagerserviceactor** implements the business logic automaton described in the main section of the problem analysis.

The QActor **trolleyactor** models the status of the trolley (idle or working) and, upon the reception of **goto(PLACE)** messages, drives the actual trolley to the specified location using a first implementation of the **RobotProxy** and of the **DirectionalPlanner**, which have already been discussed.

The QActors **parkserviceguiactor** and **parkservicestatusguiactor** are, for now, merely reactive components that print on the standard output a user-friendly version of the messages they receive. The **parkservicestatusguiactor** also sets the status of the fan upon receiving a temperature-related alarm.

The CodedQActors **weightactor**, **sonaractor** and **thermometeractor** are observable mock-objects that leverage the Java Swing components **ButtonMock** and **LedMock** (previously developed for another project) to receive user inputs and emit events based upon them. The QActor **fanactor** is based on the same concepts, but it is much simpler, because it does not need to handle user inputs. The implied taxonomy we discussed about the presence sensors is omitted at the moment.

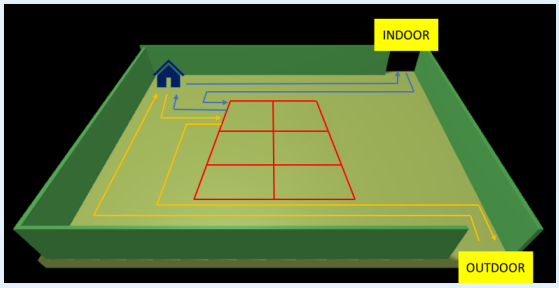
The QActors **outdoorsentinelactor** and **temperaturesentinelactor** emit their respective alarms as expected by the given requirements. The **TimerActor** is implicitly provided by the QAK meta-model with the **whenTime** keyword.

Instead of converting the format of the map to obtain a binary file, as previously discussed, we found easier to program a new rudimentary tool, called **MapUtils**, which draws a new map (with the same features as the original) and saves it in the correct format.

Testing

The executable model is compliant with the following **test plan**:

1. The transport trolley is in **home** position initially, with coordinates [0, 0, S].
2. When the weightsensor button is pressed, an "indoorOccupied" event and an enter request are sent. So, the "SLOTNUM" dispatch is sent and the **INDOOR-area**, with coordinates [6, 0, N], results occupied.
3. The transport trolley starts its path going in that cell and a "TOKENID" dispatch is sent. Then it goes on the only free **parking-slot** (the number 1, with coordinates [1, 1, E]) and comes back to **home**.
4. Then an exit request is sent and transport trolley goes in front of the **parking-slot** in which the car was parked (simulating the pick up of the car). It positions itself in front of the **OUTDOOR-area**, with coordinates [6, 4, S]. At last it comes back in its **home**.
5. If the weightsensor button is pressed while the parking-slot 1 is occupied, the transport trolley mustn't go to the INDOOR-area to take the car.



Workplan

The advancement of the project can be followed on this GitHub repository:
https://github.com/JackFantaz/BadalamentiFantazziniFinalTask2021.git

Based on what was discussed in this initial phase, we concluded that there will be needed **3 sprints** for the design of this application.

Sprint 1

- Main business logic with just one parking-slot;
- Communication with the virtual robot;
- ParkServiceGUI for the clients;

This sprint will start on **July the 19th** and we expect it to be completed within **2 working days**.

Sprint 2

- Weightsensor, outsonar and thermometer;
- Fan with manual control only;
- ParkServiceStatusGUI for the parking-manager with the buttons to start/stop the trolley disabled.

We expect this sprint to be completed within **3 working days**.

Sprint 3

- Possibility to start and stop the behavior of the trolley;
- Support for all the six parking-slots.
- OUTDOOR-area alarm and temperature alarm;
- Option to automate the control of the fan;
- Support for the real sonar device.

We expect this sprint to be completed within **3 working days** and the whole project to be finished by **July the 27th**.

Project

Sprint 1

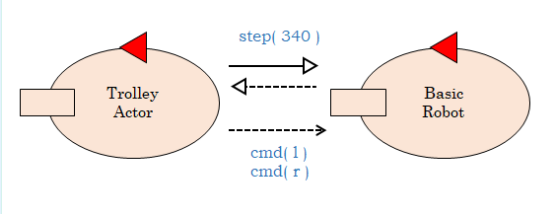
The business logic

The business logic encoded in the **parkmanagerserviceactor** follows precisely the finite-state machine diagram presented during the problem analysis, but with some simplifications given by the presence of just one parking-slot. All the states of the automaton are already present and the ability to take in and deliver meaningful **SLOTNUM** and **TOKENID** values has been arranged.

The trolley

The customer provided a new piece of software, called BasicRobot, which is compatible with the **WEnv** and with real DDR robots and requested it to be used as the implementation of the trolley.

Our **trolleyactor** does not need the presence of the **RobotProxy** component anymore and may directly send messages to the **BasicRobot** instead, which is identified as an **ExternalQActor** on a different context. The fundamental behavior of the **trolleyactor** is however unchanged.



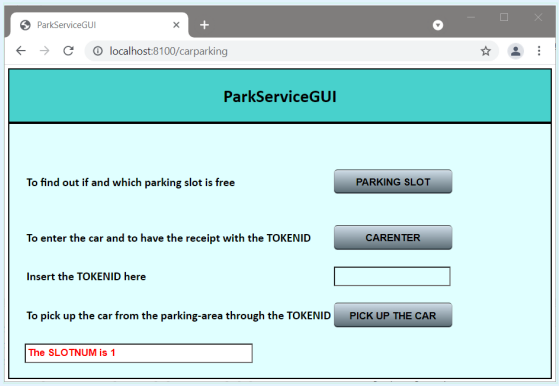
The coordinates of the points of interest of the parking-area, previously hard-coded inside the **trolleyactor**, are now externally stored as a **Prolog knowledge-base**.

```
home(0, 0, s).
parking(1, 1, e).
```

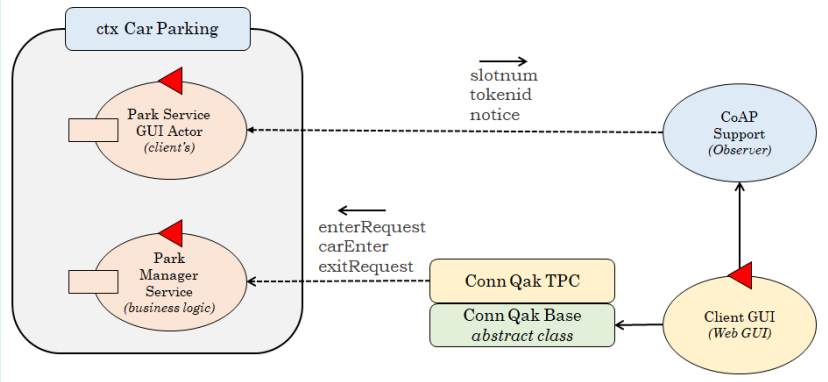
```
indoor(6, 0, n).
outdoor(6, 4, s).
```

The client GUI

The **ParkServiceGUI** for clients has been developed in **Spring Boot** as an external application and is accessible via any Web browser on port 8100.



The GUI sends messages to the business logic as an **alien component** and receives information by watching the **CoAP resource** associated with the **parkserviceguiactor**.



The messages **enterRequest** and **carEnter** and their responses (i.e. SLOTNUM and TOKENID) are **dispatches** instead of requests-replies to simplify the testing (the component making the requests and the one receiving the replies may differ) and because the external **ClientGUI** service receives status changes via CoAP.

Testing

We just need to check wether the trolley manages to reach the correct points of interest in the correct order and in a sensible amount of time. Moreover, the trolley must be unaffected by enter requests while the parking-slot is full or exit requests while it's empty. The **JUnit** module will send messages to the business logic pretending to be **parkserviceguiactor** and will try to check the reactions of the trolley.

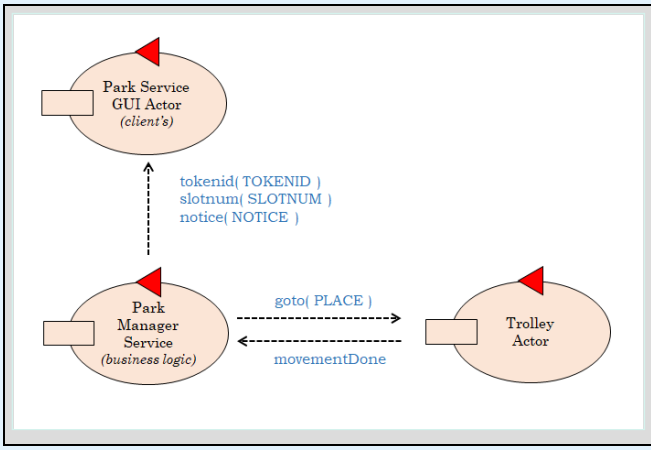
The class **DirectionalPlanner**, previously discussed and developed for the problem analysis executable model, has been refactored into a **Kotlin object** to closely resemble its wrapped component (**PlannerUtil**) and to simplify the testing. This component will be used to check the position and the orientation of the trolley.

The full test plan for this sprint can be retrieved from the following local link: ***Sprint1Test.kt*** (copy on GitHub here). We report hereunder the most significant fragments:

```
@Test
fun checkCleanSequence() {

  actor!!.forward("enterRequest", "enterRequest(0)", "parkmanagerserviceactor")
  assertNotMovingInTime(3000)

  actor!!.forward("carEnter", "carEnter(1)", "parkmanagerserviceactor")
  assertLocationInTime("6", "0", "N", 10000)
  assertLocationInTime("1", "1", "E", 10000)
  assertLocationInTime("0", "0", "S", 10000)
  assertNotMovingInTime(3000)

  actor!!.forward("exitRequest", "exitRequest(1)", "parkmanagerserviceactor")
  assertLocationInTime("1", "1", "E", 10000)
  assertLocationInTime("6", "4", "S", 10000)
  assertLocationInTime("0", "0", "S", 50000)
```

```
  assertNotMovingInTime(3000)

}
```

Summary architecture of the system as of Sprint 1



Executable model:	Testing model:
trolley.qak	*Sprint1Test.kt*

New workplan

Because of other exams we had to take, the delivery dates have shifted.

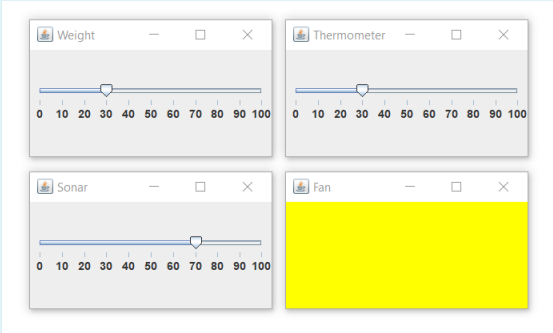
Sprint 1 will be delivered on **July the 28th**.
We expect Sprint 2 to be completed within **3 working days**.
We expect Sprint 3 to be completed within **3 more working days**.
We expect the whole project to be finished by **August the 4th**.

Sprint 2

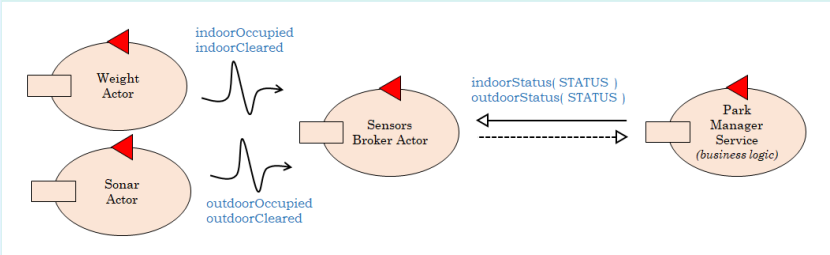
Sensors and actuators

The behavior of **weightactor**, **sonaractor**, **thermometeractor** and **fanactor** is fundamentally unchanged. We however refactored the code to decouple the actors from the mock objects and to reflect the presence sensors taxonomy presented during the requirements analysis. We also made these actors update their associated CoAP resources to perform some testing and to communicate with the Web GUIs. If requested to do so, they are also able to reply with their last issued event.

While the client and the manager GUIs should have proper Web interfaces, we believe that, for mock-objects, simpler **Java Swing** GUIs are more adequate.

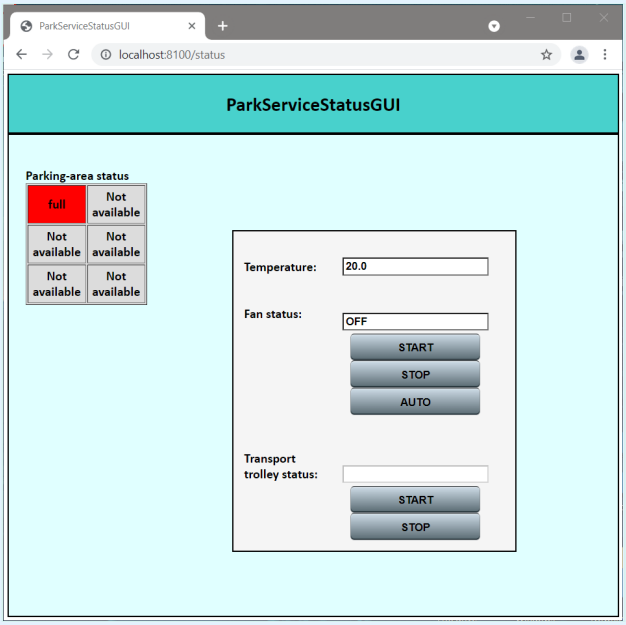


Since **parkmanagerserviceactor** has many states and the processing of the events fired by the sensors would be too cumbersome, we introduced a new actor called **sensorsbrokeractor** which reads such events and stores their meaning. **Parkmanagerserviceactor** may then **request** them on-demand when it needs to know their status.

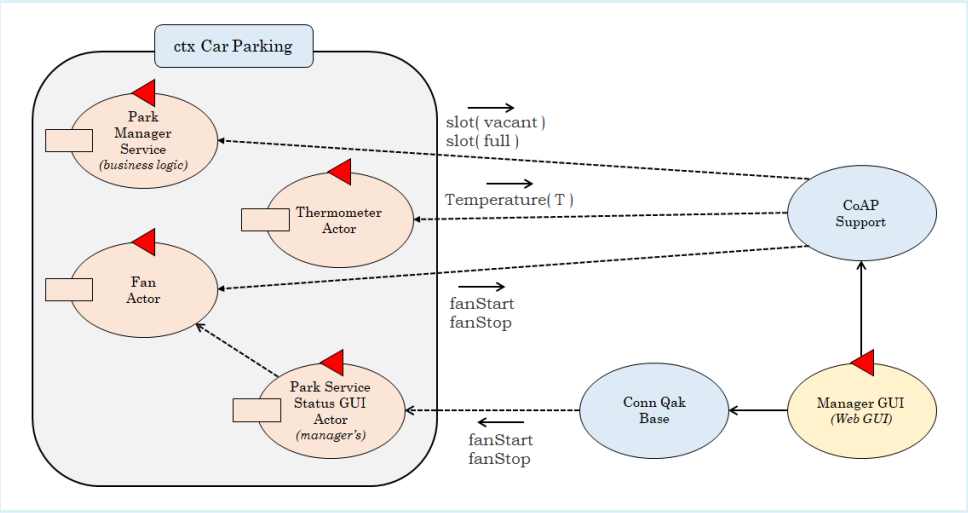


The manager GUI

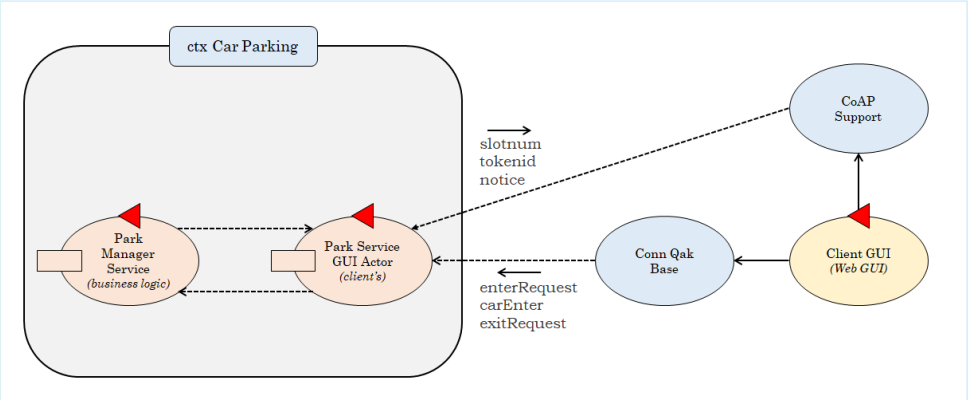
The **ParkServiceStatusGUI** for the manager has been developed in **Spring Boot** as an external application. To simplify the testing, it is currently deployed on the same node as the **ParkServiceGUI** for clients and it's accessible via Web browser on port 8100, mapping "/status".



The interaction with the business logic of this external component, called **ManagerGUI**, is essentially the same as in the other Web GUI. Although, to receive data without waiting for user input, we regularly update the page via **AJAX** and read each interested CoAP resource. The **parkservicestatusguiactor** receives messages from the external GUI and redirects them to the correct actor.



The **ClientGUI** has been refactored to follow the same cleaner approach. Since all the data needed by the **ClientGUI** come from messages specifically intended for the **parkserviceguiactor**, it does not need to interact with any other actor inside our context.



Testing

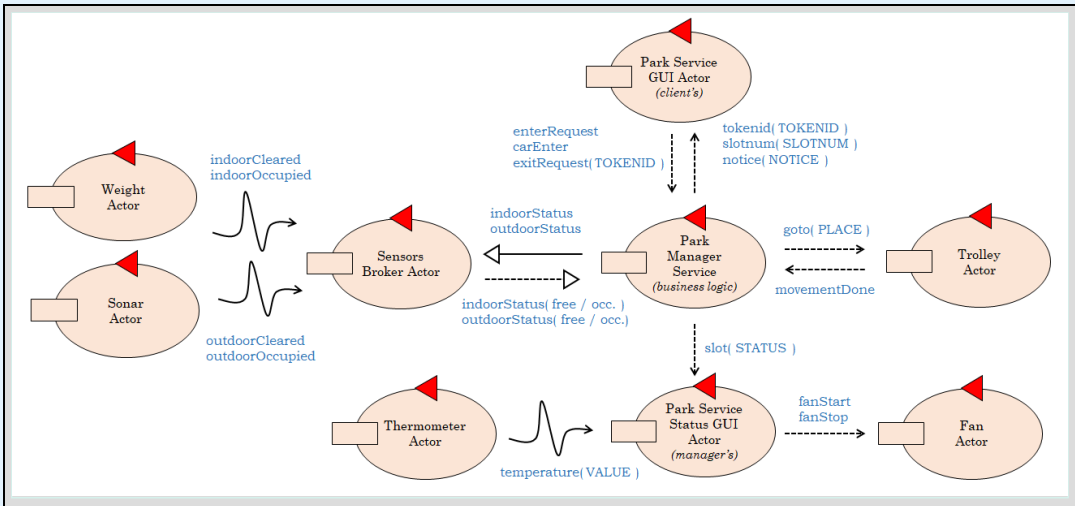
All the tests conceived for Sprint 1 must still hold. Moreover, we defined a new test case to ensure that the system checks **INDOOR-area** and **OUTDOOR-area** to be free before starting related actions.

We also implemented an additional test case to check wether the mock objects are working as intended. Such test requires **human interaction** to be completed, otherwise we would not be able to assure that the GUIs are working. All the other test cases simulate the behavior of the mock objects and do not require human interaction to be completed.

We report hereunder the most significant fragments of the new tests:

```
@Test
fun checkDoors() {

  actor!!.emit("indoorOccupied", "indoorOccupied(0)")
  actor!!.forward("enterRequest", "enterRequest(0)", "parkmanagerserviceactor")
  actor!!.forward("carEnter", "carEnter(1)", "parkmanagerserviceactor")
  assertNotMovingInTime(3000)

  actor!!.emit("indoorCleared", "indoorCleared(0)")
  actor!!.forward("enterRequest", "enterRequest(0)", "parkmanagerserviceactor")
  assertLocationInTime("6", "0", "N", 10000)
  assertLocationInTime("0", "0", "S", 20000)

  actor!!.emit("outdoorOccupied", "outdoorOccupied(0)")
  actor!!.forward("exitRequest", "exitRequest(0)", "parkmanagerserviceactor")
  assertNotMovingInTime(3000)

  actor!!.emit("outdoorCleared", "outdoorCleared(0)")
  actor!!.forward("exitRequest", "exitRequest(0)", "parkmanagerserviceactor")
  assertLocationInTime("6", "4", "S", 20000)
  assertLocationInTime("0", "0", "S", 50000)

}
```

Summary architecture of the system as of Sprint 2



Executable model:	Testing model:
trolley.qak	*Sprint2Test.kt*

By: Giacomo Fantazzini and Claudia Badalamenti
Email: giacomo.fantazzini2@studio.unibo.it - claudia.badalamenti@studio.unibo.it