Requirement Analysis

Setting

- The **teaRoom** is described as a rectangular room composed of:
    - **N teaTable**: tables placed inside the tearoom, where the admitted **Client** can consume his tea.
    - **serviceArea**, composed of:
        - **serviceDesk**: where the entity barman prepares the tea after receiving a request by the waiter;
        - **home**: where the waiter can rest if it has no tasks to do.
    - **hall**: where, when he arrives, a **Client** has to wait here before entering the teaRoom (client behaviour explained in point 2). It is equipped with:
        - **a presenceDetector,** which can detect the presence of a person or other entity;
        - **a smartbell,** which measures the temperature of the Client that wants to enter the tearoom and, if the Client's temperature is less than 37.5˚, ***sends a request message*** *to the waiter*.
        - an **entranceDoor**, through which the Client will be admitted inside the tearoom;
        - an **exitDoor**, through which the Client can leave the tearoom
- **clientIdentifier:** value to univocally represent a client's request of entering the tearoom. It is assigned by the smartbell to the client and it is given to the waiter with the aforementioned request.
- A **tearoom is considered safe** if there are no people inside with a temperature greater or equal than 37.5˚ and if there are clean tea-tables posed at a proper distance.

Client

- **notifiy interest in entering the tearoom**: once in the **hall**, the Client ***has to send a notification*** to the **smartbell**, whose behavior has been described in point [1.A.iii.2].
- ***maxWaitTime***: time value that will be given by the **waiter** if there are no free and clean tables available and he's not been sent away because of the temperature, after which either he has entered, or he has to leave.
- ***maxStayTime***: maximum time that the Client can spend at a teaTable. After it expires, the client has to leave, no matter if he's finished the tea or not.

Waiter

- The **waiter tasks**, listed in the requirements, are a set of actions the waiter should be able to perform, one at the time, *optimizing as much as possible the execution so that the waiting time of the requests coming from each client is minimized*.
- **Convoy the Client**: once a Client is free to enter the tearoom or ready to leave it, the waiter has to accompany them to and from the table, from and to the entranceDoor and exitDoor, respectively.
- **Clean the (tea)table**: once the client leaves the table, the waiter has to clean it before another client can occupy it.
- The waiter is also a ***Robot***. This means that there needs to be a system to interact with the robot hardware to make it move around the room.
- Barman
    - ***receive* order from waiter**: the orders are *transmitted through a WIFI* device by the waiter to the barman.
    - ***notify* waiter of drink ready**: the barman has to send a notification to the waiter once the drink order by a Client is ready. The drink should be prepared in a time that is significantly smaller than the client's maxStayTime.

Entities as Actors

What we can infer from the requirements is that:

- we have identified different entities that will come into play in the system (barman, waiter, client, smartbell);
- all of these entities have a behavior that can be represented as a **Finite State Machine** while also have to keep a **readable state** for the manager;
- they will need to interact with each other through different kinds of messages.

These three points have highlighted the need for a model representation through the concept of Actor as a Finite State Machine. To do so, we introduce the QAktor meta-modeling language, referenced here, so as to close the abstraction gap as much as possible from the beginning and also develop working prototypes fast and easily.

Defining the states

- During this first analysis, we have identified the main states the various actors can be in:
    - **Waiter** can be answering the client's request to enter, deploying the client, cleaning the table, moving or waiting for a new task to execute;
    - **Barman** can be preparing the beverage or waiting for a new order;
    - **Client** can be Outside, waitingInTheHall, seated, finished, payed and left;
    - **SmartBell** can be evaluating a client or free.
- Even though teaTable is not an actor, we need to keep its state memorized for the system to work properly and for the current state of the teaRoom to be complete. The states the teaTable can be in are:
    - ***tableClean***: without food residue and sanitized and available;
    - ***tableDirty***: not clean but available;
    - ***tableBusy***: occupied by a client.

Interaction Diagrams

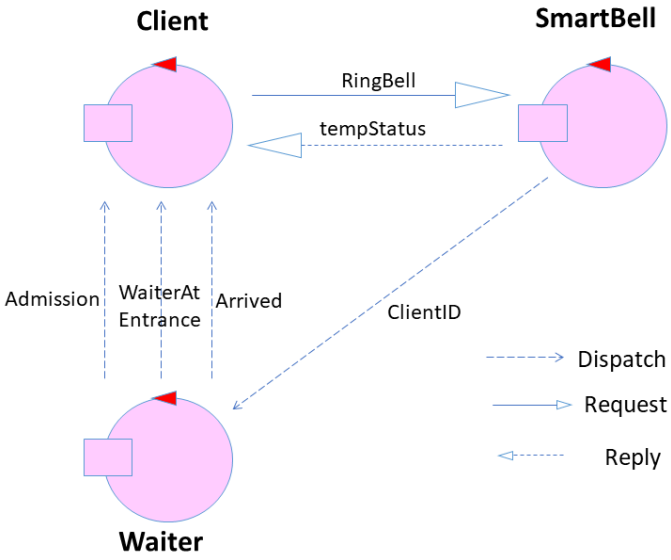

Figura 1: Client arrival

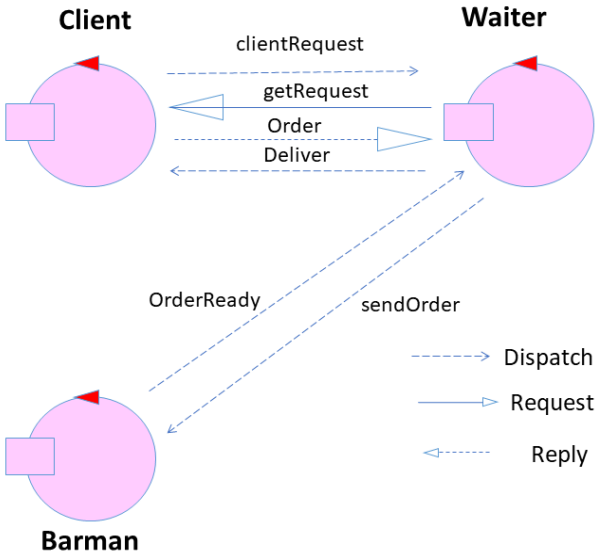

Figura 2: Client orders

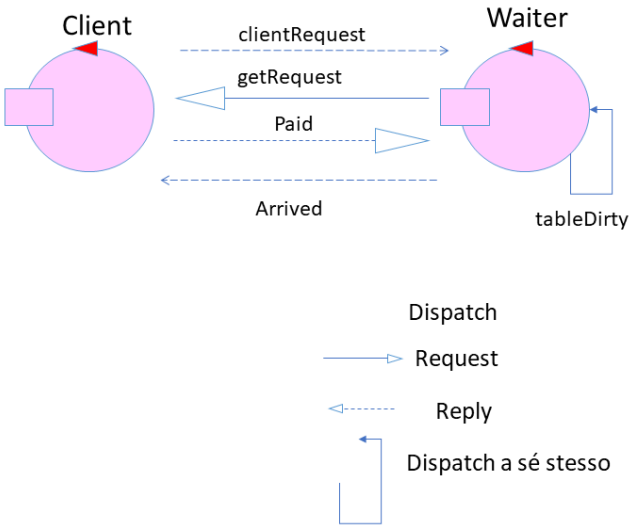

Figura 3: Client pays

State Diagrams

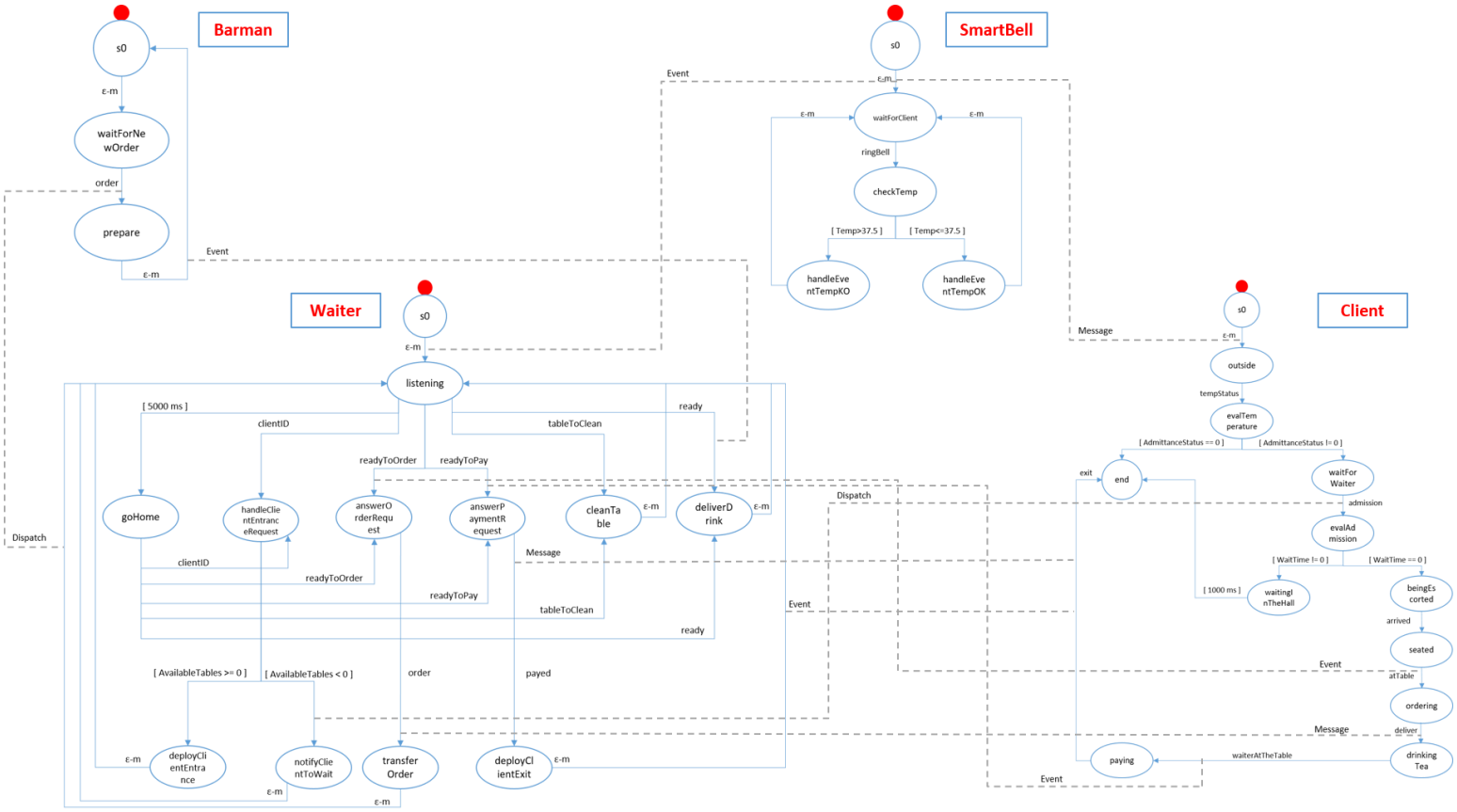

Figura 4: State diagram

Test Plan

We plan to test various activities:

- The **smartBell** must let in the **hall** only **clients** that have a temperature below 37.5° Celsius.
- When a client enters the tearoom, he may only sit at a table that is in the state **tableClean**. If there is no such table available, then the **waiter** must inform the **client** about the **maximum waiting time**.
- The client must leave the **hall** when the **maximum waiting time** is over.
- The client must receive the drink he ordered.
- The client must **pay and leave when the maximum stay time** is over.

Problem Analysis

System Architecture

As already pointed out in the Requirements Analysis, the system is composed of the following entities:

- **Client**: the “external” source of input for the system;
- **Manager**: needs to be able to read the system’s current state through a WebApp;
- **Waiter**: actor that moves around the tearoom to execute the tasks required of him by the current state of the tearoom and the requests of the Client;
- **Smartbell**: actor that manages the entrance of the tearoom;
- **Barman**: actor that manages the orders coming from the Waiter.

Initial hypothesis

To simplify this first analysis of the system we shall make some hypothesis:

- At the moment we will consider only one client at the time;
- We will not consider the existence of the manager and the web application he would use to monitor the state of the tearoom;
- we will not consider the tableWe will not take into account optimization of tasks.

Implementing the Waiter

As it has been pointed out in the requirement analysis, the waiter is supposed to be a DDR Robot. This means that the waiter has to be decomposed in multiple actors to model the different facets of the tasks it has to accomplish, in particular there is a need to separate the Actor interacting with the actual Robot, telling it where to go, and the “mind”, which should implement the logic of the actions.

We can thus identify the following actors that come into play:

- **Waiter**: Deals with the high level logic of the application such as the interaction with Client, Barman and Smartbell but also choosing when to do which task.
- **WaiterWalker**: Selects a set of commands to send to the basic robot to reach a certain destination.
- **BasicRobot**: Receives a command and makes the Robot execute it.

The infrastructure that handles the BasicRobot has already been implemented to solve another client’s problem (it.unibo.quak20.basicrobot doc). This actor already implements all the necessary commands, so there is no need to modify it.

The WaiterWalker and the Waiter are, instead, to be developed.

WaiterWalker

Knowledge of the Environment

An issue that emerges from the Requirement Analysis is that the WaiterWalker has to be able to "know" the room and plan how to get from point A to point B, possibly optimising its route.

To address this issue, the best choice is to implement a component whose job is to keep track of the topography of the room, compute the best sequence of moves to reach the position requested and interact with the basicrobot to actuate those moves: the WaiterWalker.

Since we also have already solved the planning problem, we can exploit the library `it.unibo.planner20-1.0`. The documentation of this library can be found here. This tool provides us with operations for creating plans, managing plans as action sequences, inspecting robot position and direction and managing the room-map.

As clearly stated in the aforementioned documentation, we know that, since the basicorobot handles movements in **steps** (where a step is a movement that moves the robot by its size in one of the four cardinal directions: Up, Down, Left, Right), to represent the tearoom logically we can imagine it as a rectangle divided in squares the size of the robot.

Since we are already given the topology of the tearoom, we can already write the map file to use with the library, which can be found in the file `tearoomExplored.txt` in the prototype.

Waiter

The outline of the Waiter behaviour has already been implemented in the Requirement Analysis prototype.

The only thing the Waiter should do is handle the logic of executing the tasks. What we need now is to add the interaction with the WaiterWalker, to tell it what kind of movement it has to make.

Interaction between Waiter and WaiterWalker

Since we want to keep the specifics of implementation separated between the Waiter and WaiterWalker, so as to reduce dependency as much as possible, we want to communicate to the WaiterWalker *only what kind of task needs to be performed* (e.g. executing task: deploying client) and leaving the implementation of *how* to reach point B from current point A completely to the Walker.

This calls for a new exchange of messages, which can be represented with the following diagram:

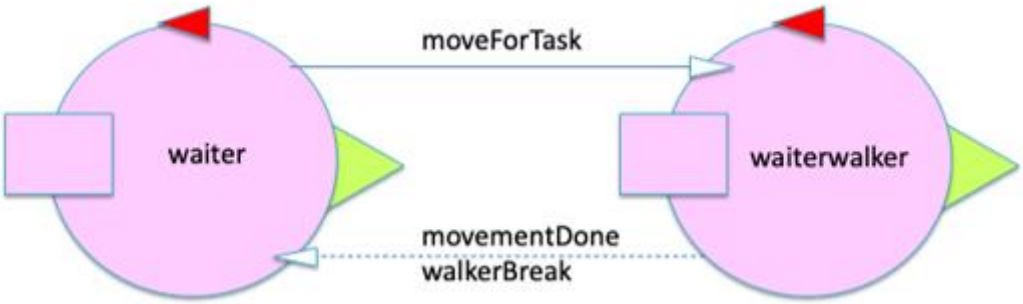

Figura 5: : Interaction between Waiter and WaiterWalker

Where the Waiter uses the payload variable TASK to notify exactly what task has to be performed and N as a “jolly” payload to handle multiple positions for the same kind of task (like the barman or the teatable).

Obviously, what goes inside the TASK payload variable should have a correspondence between the Waiter and the WaiterWalker. The latter should, in particular, keep a mapping between squares on the map that represent the goal of the task and the ID of the task itself.

Building a Knowledge Base

With an eye on what we’ll have to deal with later, we can see how the fact that we need to keep and eventually retrieve so much information calls for the introduction of a *Knowledge Base* (also referred to as KB). This will need to store:

- the current state of the tables;
- the matching between task IDs and map positions;
- the current state of the waiter.

The implementation details are left to the Project section.

A first Prototype

To properly show how the Application should work at this stage, we have developed a first prototype.

Messages

As we have already explained, the actors need messages to interact. As visible in the next paragraph, we use different kinds of messages depending on the exchange required. Here is a summary of the messages used:

- When the *Client* arrives, she sends a Request ringBell, specifying her temperature in the `TEMP` variable, to the *Smartbell* and waits for the Reply tempStatus, which contains a variable `STATUS`, that is 0 if the temperature is too high to enter the tearoom or 1 if it's ok. The variable `ClientID` contains the *clientidentifier* to identify the client.

```
Request ringBell : ringBell(TEMP)
Reply tempStatus : tempStatus(STATUS, CLIENTID)
```

- If the Client's temperature is fine, the *Smartbell* sends a Dispatch message to notify that there is a client waiting to be admitted.

```
Dispatch clientID : clientID (CLIENTID)
```

- Once the *Waiter* handles the previous Dispatch, it sends a Dispatch message to the *Client* to let them know how long they have to wait through the variable `MAXWAITTIME`. If 0, the *Waiter* is moving to take them to the table right away.

```
Dispatch admission : admission(MAXWAITTIME, CLIENTID)
```

- When the *Waiter* arrives at the entrance, he sends a Dispatch message so that the *Client* knows he's there. This is when the deployment to the table begins.

```
Dispatch waiterAtEntrance : waiterAtEntrance(OK)
```

- Once the *Client* is ready to order the tea, she sends a Dispatch message and waits for the *Waiter* to come at the table.

```
Dispatch readyToOrder: readyToOrder(TABLE, CLIENTID)
```

- As soon as the *Waiter* reaches the table, it requests the order to the *Client*, which Replies with what kind of tea she wants

```
Request getOrder : getOrder(TABLE, CLIENTID)
Reply  order : order(TEA)
```

- When the *Client* Replies with the desired tea, the *Waiter* sends a Dispatch message to the Barman.

```
Dispatch sendOrder : sendOrder(TEA, TABLE)
```

- When the *Barman* has done preparing the tea, he sends a Dispatch message to notify it

```
Dispatch orderReady : orderReady(TEA, TABLE)
```

- When the Tea is ready the *Waiter* notifies the *Client* and brings it to him.

```
Dispatch deliver : deliver (TEA)
```

- When the *Client* is ready to pay, he sends a Dispatch message to notify it

```
Dispatch readyToPay : readyToPay(TABLE, CLIENTID)
```

- As soon as he receives the Dispatch message notifying that the Client's ready to pay, he goes to the table, sends a Request for the money owed in the variable MONEY to the I and it answers with a Reply to complete it.

```
Request pay : pay(MONEY, CLIENTID)
Reply  paid : paid(MONEY)
```

- When the *Client* has done the payment, the *Waiter* deploys her to the ExitDoor. To notify the *Client* that they have arrived a Dispatch message is sent.

```
Dispatch exit : exit(OK)
```

- Once the *Client* has paid, the table becomes dirty and an “auto-Event” is raised for the *Waiter*, so that he can clean it. The number of the table dirty is in the payload variable N.

```
Dispatch tableDirty : tableDirty(N)
```

Test Plan

Other than what has been previously outlined in the requirements analysis the to run in this phase regard the implementation of the waiter walker and its ability to plan the moves to take travel across the room.

Project

By using QAk actors in the analysis phase we now have an executable kotlin model.

To deal with the message exchange between actors we have used the Message Queuing Telemetry Transport protocol (MQTT) as it is a message-subscribe model well suited to iot applications. In particular, we used mosquitto as a message broker as it is a lightweight open source solution. https://mosquitto.org/

Implementing the Knowledge Base in Prolog

We introduce a *knowledge base* (also referred to as KB) written in **Prolog**, which will be stored in a file and loaded at startup. This way, it will be easy to declare and modify the state and write rules to retrieve information about it. The usage of Prolog results as the best choice due to the fact that the qak library already has a TuProlog engine embedded in each Actor.

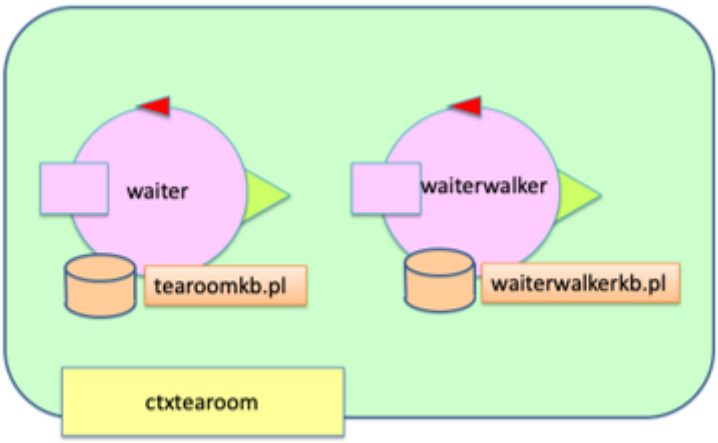

Figura 6

The KB loaded in the WaiterWalker will have to keep, as we said, all the facts that tell it where to go depending on the TASK received.

Since we also need to keep track of the state of the tables, to correctly assign a free one to the incoming client or let the Waiter execute the cleaning task, and the state of the overall tearoom, we introduce a KB for the Waiter as well.

Architecture Diagram

Finally, we can see how the architecture we are creating is a layered architecture.

For example, this diagram shows how the waiter interacts with the client's requests and the various layers involved.

- **Client**:
  - Sends messages to the waiter with a request
- **Waiter**
  - Receives requests from the client and sends a dispatch to the waiterWalker to deal with the movement.
- **WaiterWalker**
  - Receives a certain task and determines where to go to accomplish that task and which route to take to reach that location using prolog and a knowledge base.
  - Sends the basic move commands for each step to the basic robot.
  - sends a dispatch to signal to the Waiter when the movement is completed or if there has been an error.
- **Basic robot**:
  - Is able to execute basic move commands, by handling a dispatch (for the elementary commands w | s | ...)

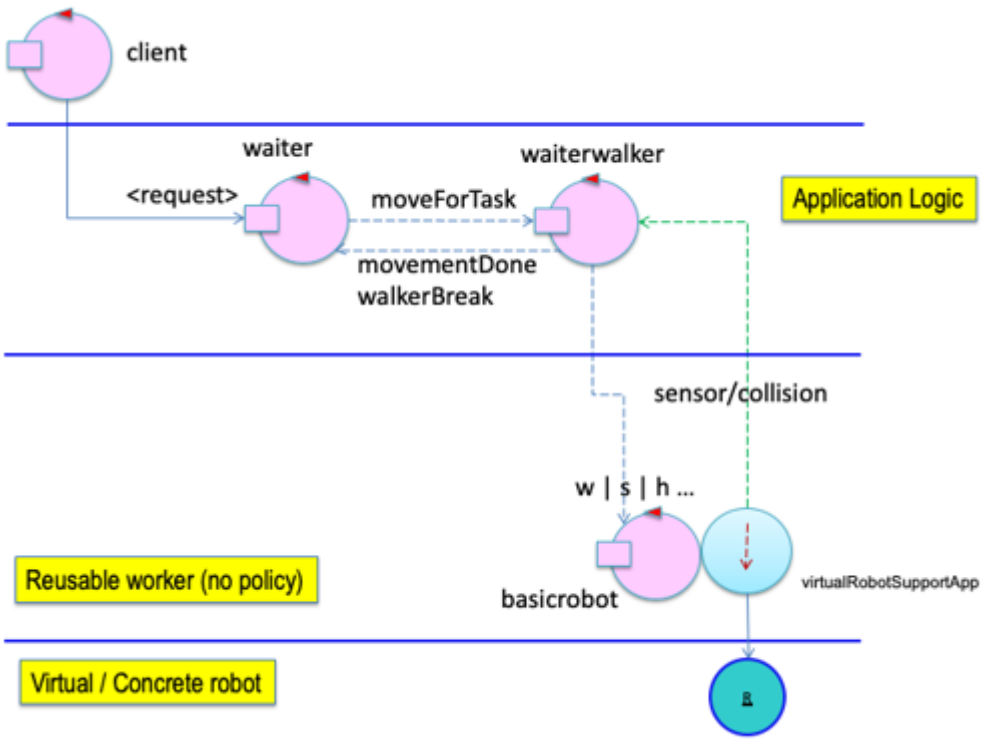

Figura 7

There are 2 more *Actors* not in the diagram:

- **SmartBell**:
    - Receives a request to enter from the client and if the temperature is less than 37.5 it will send a confirmation to the client and send a dispatch to the waiter.

- **Barman**:
    - Receives a dispatch from the waiter when there is a drink to prepare and sends him a dispatch when it's ready.

To add to Backlog

In future sprints we plan to add another actor between waiterwalker and the basicrobot to act as a middleware that will receive the target square to reach and deal with the planning of the route and single elemantary commands to the basic robot.

Test Plans

The testing has been done using JUnit 4. Other than the tests outlined in the requirement analysis and problem analysis we have focused on testing that the virtualRobot's position is correct in each state. The test should focus on the correctness of:

- Waiter position when taking an order, taking a drink from the barman and bringing a drink to the client;
- Waiter position when convoying a Client to the table and to the exitDoor;
- Waiter position when no tasks are pending.

The tests can be found at iearoom.SPRINT1.test