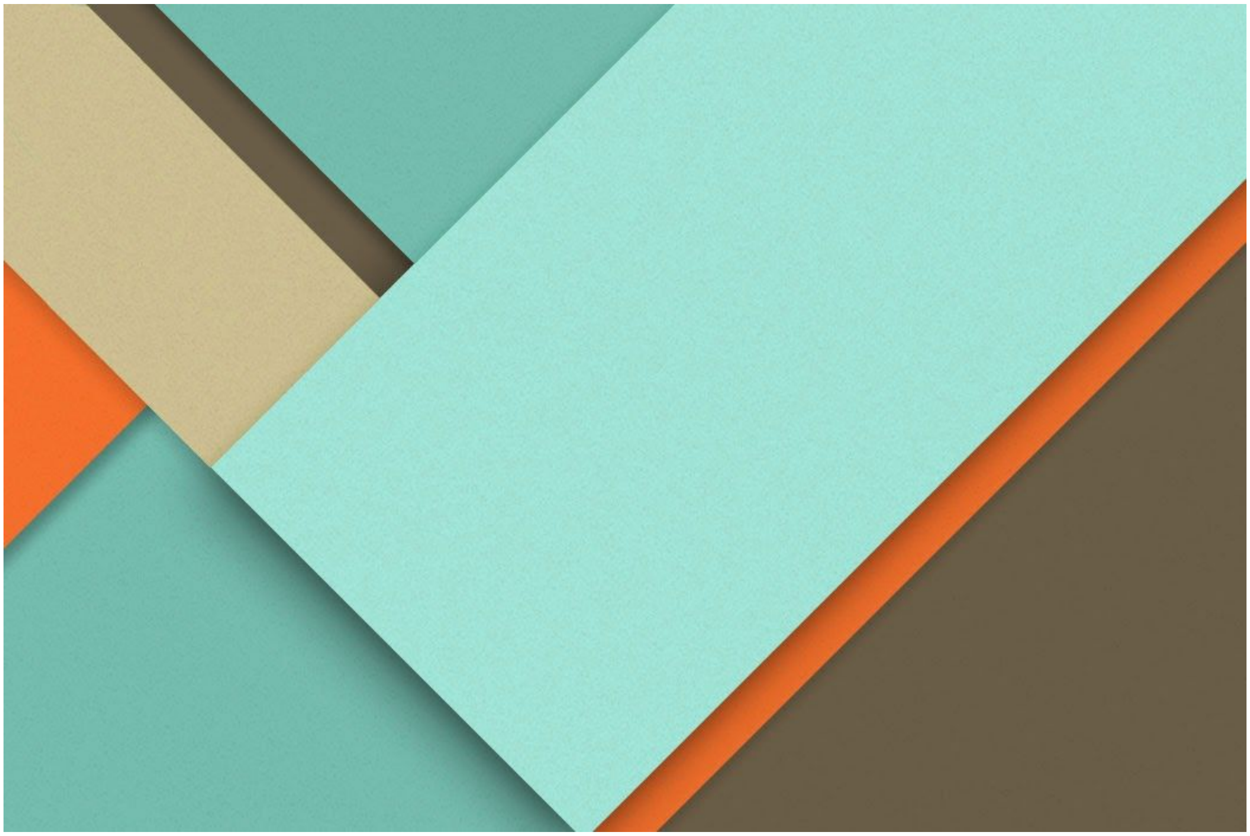

SPRINT1

10.09.2020

Laura Mazzuca, Nicholas Antonio Carroll, Giuseppe Giorgio

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.
- 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

- **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.
- **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 the waiter**: the orders are transmitted through a WIFI device by the waiter to the barman.

● **notify the waiter that a drink is 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.

A closer look at the entities in play

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;
- the system will be distributed, and the entities will need to interact with each other through different kinds of messages.

Aiming to close the Abstraction Gap

Since we want to show our commissioner a preview of how the system should work, we want to start prototyping from this early stage. Doing so requires a tool that can capture the intrinsic nature of the entities we have pointed out in the previous paragraph and translate it into runnable code as fast as possible.

Thankfully, our software house already provides us with a meta-modeling language, [referenced here] and called QAk, which gives us the possibility to easily describe the essence of the behavior the system should have.

Defining the Tearoom State

As per the Manager User Story, we know we want to keep the information about the state of the tearoom. But what should it comprehend? We have identified the following information:

- Waiter position;
- Waiter current task;
- Tables state (dirty, clean, busy).

Prototype

To give a first example of how the system should behave, at least with one client, we have built a message-driven prototype, which can be found in the project `tearoom.model`, inside the folder called RequirementAnalysis.

The following diagrams give a quick idea of how the system has been modeled at this stage:

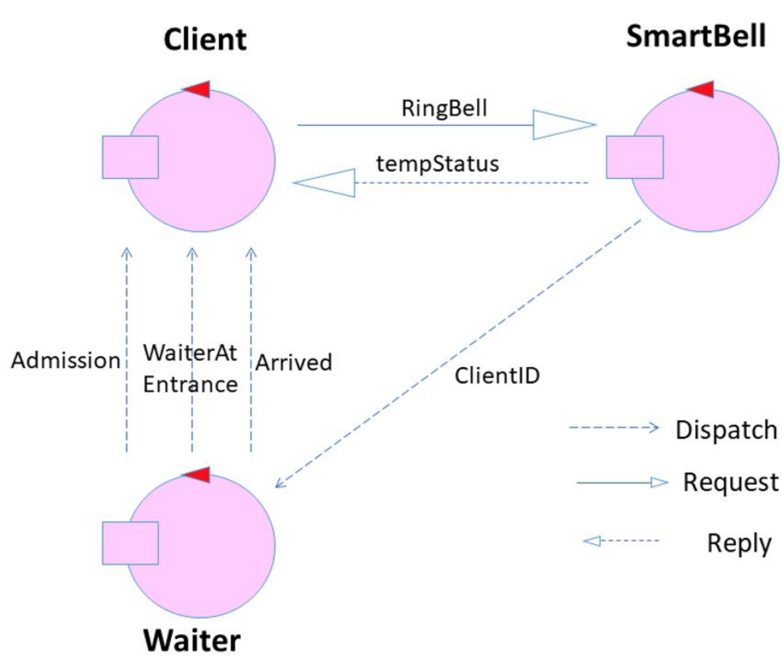

FIGURE 1 CLIENT ARRIVAL

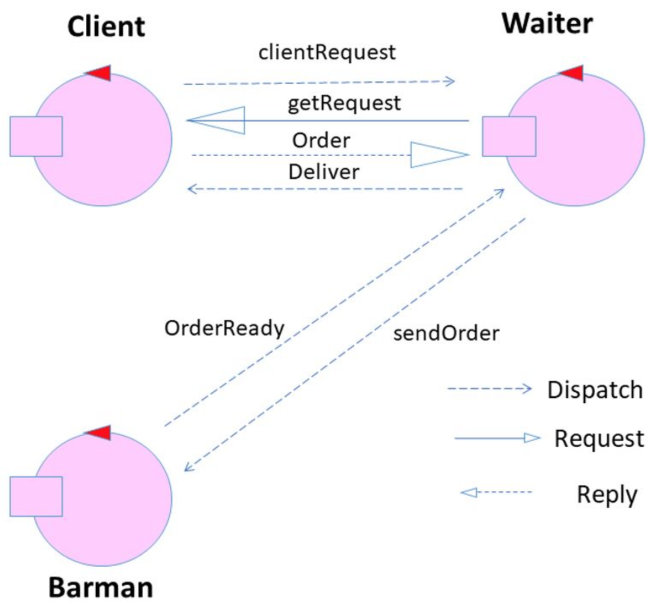

FIGURE 2 CLIENT ORDERS

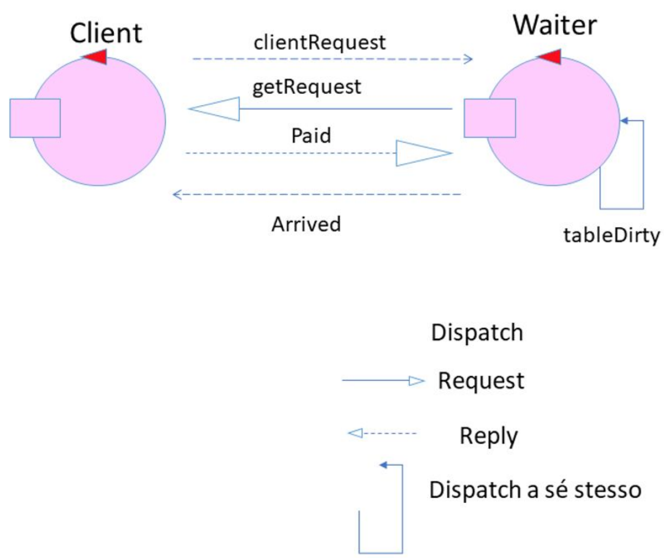

FIGURE 3 CLIENT PAYS

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

Assumptions

To easily tackle all the requirements SPRINT after SPRINT, we will simplify this first analysis of the system by making these assumptions:

- we **will** consider the presence of **only one client**;
- The client, the barman and the smartbell will be simulated and grouped together with the waiter context-wise;
- 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 table's states;
- we **will not** take into account optimization of tasks;
- The waiter **won't** be able to interrupt a task;
- We **won't** take into account the MaxStayTime and MaxWaitTime timeouts.

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.

It's immediately clear how this system will, in the end, be distributed: the Waiter is working on a Robot, while the Client and Manager's interface will most likely mork on a Server; the Barman is working on "a device connected through WiFi", as the Requirement Analysis states, and the Smartbell will most likely be deployed on a temperature-checking device.

Having noticed this since the beginning, we can start planning our development accordingly. In particular, this kind of topology induces us to implement a **Microservices Architecture**, where each Entity is decoupled from the other and the communication between them will take place through a remote access protocol.

Using QAktors will greatly aid us in developing such a system, thanks to the features already present in their implementation, such as the possibility to communicate through Mqtt or CoAP, as well as the standard TCP connection.

Contexts

Another perk of QAktors is the possibility to specify on which Context (computational node) the Actor is active.

This brings us to the first step of tackling our problem: what context should be given to each Actor?

In this first SPRINT we will only consider a single Context: `ctxtearoom`. This Context will contain all the Actors supposed to interact with the Waiter, so the Client, the Barman and the Smartbell. This will give us a much more easy system to build so we can approach the first steps of the problem in a more "controlled" way.

Instead, the basicrobot, which will be mentioned in the following paragraph, will be considered right away on a different node, called `ctxbasicrobot`.

Implementing the Waiter

As it has been pointed out in the requirement analysis, the Waiter is supposed to be a Robot. This means that its behavior should 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, so as to keep the code both reusable and compliant to the Single Responsibility Principle.

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 by our software house 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 our software house has 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's folder `tearoom.domain`.

Waiter

The outline of the Waiter behavior 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 the Waiter and the 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:

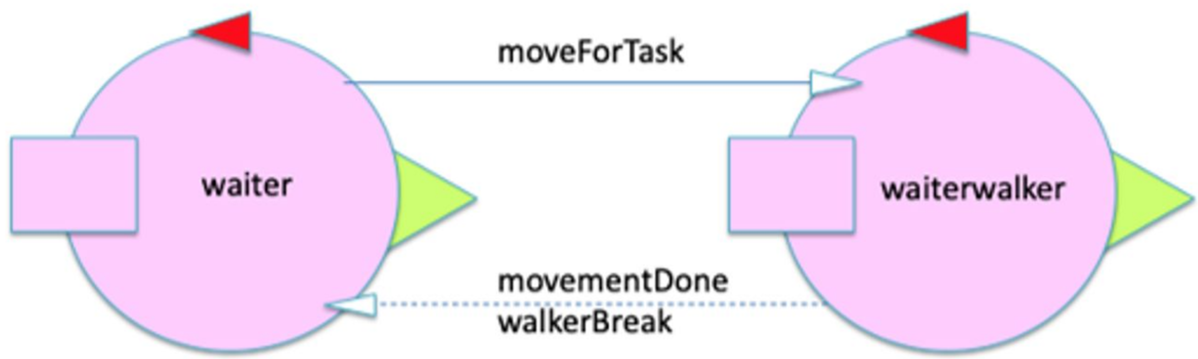

FIGURE 4 INTERACTION BETWEEN WAITER AND WAITERWALKER

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 what we'll have to deal with later in mind, as it has already been explained in the Requirement Analysis that we have to keep a consistent state of the Tearoom to show to the Manager, 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 current state of the waiter;
- the matching between task IDs and map positions.

Clearly, we need to ensure consistency, so the implementation should keep that in mind when choosing how to handle the storage and access to it.

The Prototype

To properly show how the Application should work at this stage, we have developed a prototype built on what was done during the requirement analysis.

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 tests to run in this phase are:

- the waiterwalker's ability to plan the moves to take travel across the room;
- the waiter's ability to handle the client's requests.

Project

By using QAk actors in the analysis phase we now have an executable kotlin model, so we will be building on top of that to keep the vantage gained on the abstraction gap and the direct modeling of the system into a Microservices architecture.

Waiterwalker and Basicrobot interaction

To deal with the message exchange between these two 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.

This choice has been made since we foresee the deployment of the basicrobot to be, most likely, divided by the deployment of the Waiterwalker and the Waiter, which could, instead, work on the same node.

Implementing the Knowledge Base

Since the need for a Knowledge Base has been highlighted during the Problem Analysis, we now approach the implementation focusing not only on the support we will use, but also on consistency.

Prolog

Given that each QAktor contains itself a tuProlog engine, we can exploit its declarative style programming language while also ensuring that only the Actor assigned to handle the KB can access it or modify it. This can **ensure a one point of access**, which will need to be built to handle correctly every event that could modify the state.

Waiter as a Single Point of Access

We've seen how it's the Waiter that handles any request of changes that happen in the tearoom, so we want it to be the handler of any change in the state. This means that the KB handling the Tearoom State will be concentrated in the Waiter as a single point of access. To make it possible for other future applications to receive updates on what the state actually is, we also make the Waiter a CoAP resource.

Waiterwalker

We also need something to keep a mapping between the positions to reach and some IDs given to the TASKs the waiter can perform, moving to the corresponding position.

To do so, we also give the WaiterWalker a Knowledge Base, which contains the mapping.

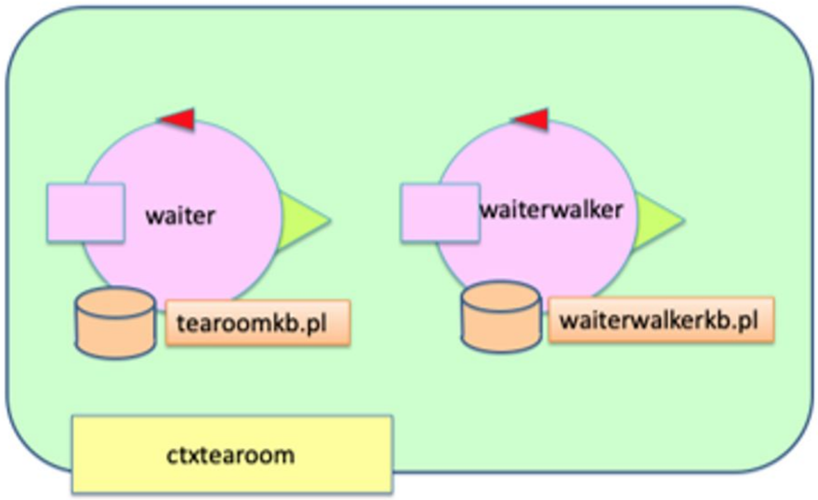

FIGURE 5 DIAGRAM OF KB DISTRIBUTION

Architecture Diagram

Finally, we can see how the architecture we are creating for the Waiter is a layered architecture based on Microservices.

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.
- Basicrobot:
  - Is able to execute basic move commands, by handling a dispatch (for the elementary commands w | s | ...)

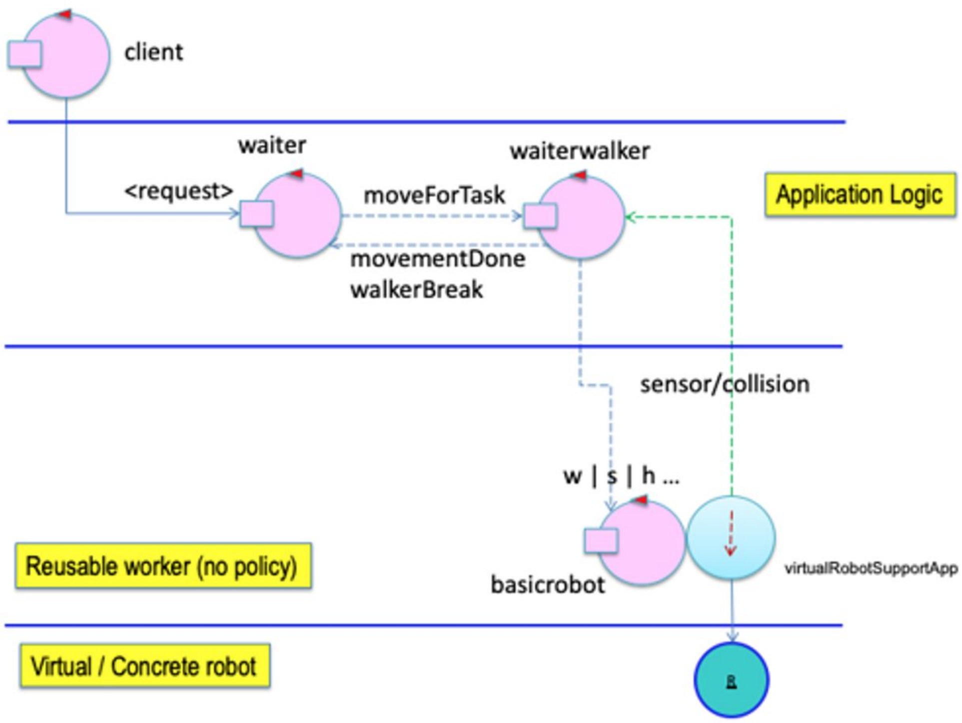

FIGURE 6 LAYERED ARCHITECTURE DIAGRAM OF THE WAITER

There are 2 more Actors not shown 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.

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 deploying a Client to the table and to the exitDoor;
- Waiter position when no tasks are pending.

The tests can be found at `tearoom.SPRINT1.test`.

Backlog Update

In the light of this SPRINT development, we plan to add another actor between the 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 elementary commands to the basic robot.