Lab ISS 2021 | iss0

Introduction

This case-study starts to deal with the design and development of proactive/reactive software systems that use aynchronous exchange of information.

Requirements

Design and build a software system that allow the robot described in *VirtualRobot2021.html* to exibit the following behaviour:

- the robot lives in a rectangular room, delimited by walls that includes one or more devices (e.g. sonar) able to detect the presence of obstacles, including the robot itself;
- the robot has a den for refuge, located near a wall;
- the robot works as an explorer of the room. Starting from its den,the goal of the robot is to create a map of the room that records the position of the fixed obstacles. The presence of mobile obstacles in the room is (at the moment) excluded;
- since the robot is 'cautious', it returns immediately to the den as soon as it finds an obstacle. it also stops for a while(e.g 2 seconds), when if 'feel' that the sonar has detected its presence.

Requirement analysis

Analysis already defined in the previous document 'Corsaro_Lucrezia_ce.pdf'.
In addition we can clarify:

- **feels**: the robot receives a signal sent by the sonar that has detected its presence;

Verification of expected results (Test plans)

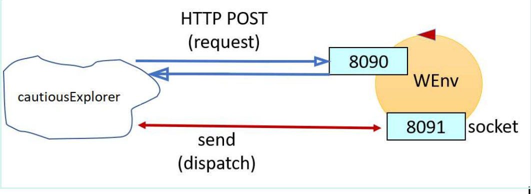
It is necessary to verify that the robot has moved from the den, has found an obstacle, and has returned to the den.	The verification must be carried out via software, without the need for intervention. of a human user.

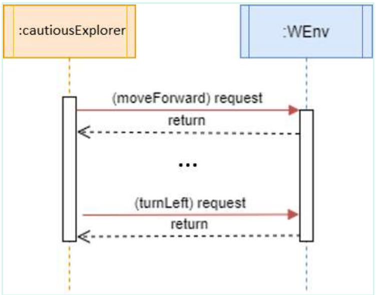
Problem analysis

Relevant aspects

1. Create a distributed system consisting of two macro-components:
   - the (virtual) robot supplied by the client
   - our application (cautiousExplorer) which sends commands to the robot with a request-response pattern in order to satisfy the requirements
2. The robot can be controlled via the network in two different ways, as described in VirtualRobot2021.html: commands:
   - sending messages to port 8090 with HTTP POST protocol
   - sending messages to port 8091 using a websocket
3. Since there are numerous libraries in many programming languages that allow the sending of such commands, no significant abstraction-gap is identified on an operational level.
4. We estimate that a first prototype of the application should be able to be built in seven working days (at most).

Logical architecture

 HTTP POST (request) 8090 WEnv cautiousExplorer 8091 socket send (dispatch)	The exact nature of the cautiousExplorer component will be defined in the design phase. Regarding the interaction we can say that: - the use of the HTTP protocol seems completely adequate, at least in a first phase; - the use of the websocket could prove to be more flexible (as it allows to receive information emitted by WEnv in a 'spontaneous' way) and more efficient (as it reduces the protocol hierarchy).
Another model (UML)	This model 'abstracts' from the technological details of the interaction.



MAP

We can build two possible different types of map:

- a string that represents the robot path expressed as a sequence of moves.
- the room is divided into cells of the size of the robot, in order to build a map as in the example:

  |r,0,0,0,0,
  |1,0,0,0,0,
  |1,x,0,0,0,
  |1,0,0,0,0,

  In this rappresentation, we suppose that:
  - r means: cell occupied by the robot
  - 0 means: cell non explored
  - 1 means: cell explored
  - x means: cell with obstacles

Problems identified

The robot to find the obstacle can perform a series of both random and organized moves. It is planned to send moves in an organized way to avoid repetitive loops or paths. The robot returns to the den retracing the moves in the opposite direction, to avoid encountering other obstacles on the way back.

1. **Interaction abstraction**

   The specification of the exact 'nature' of our CautiousExplorer software is left to the designer. However, we can say here that is it not a database, or a function or an object. The software system should be made as independent as possible from the communication protocol used for the interaction with WEnv.

2. **Testing**

   We can prefigure that the solution of the problem consists of the following algorithm:

```
let us define emum direction {UP,DOWN,LEFT,RIGHT}
the robot starts from the den
until the robot hits an obstacle:
        sends the robot a request to execute random commands, continue to do it, until the answer of a command becomes fa
        if the robot receives the signal from the sonar, it stops for two seconds;
        if the robot receives the sonar signal several times, it alternates a movement with a two-second wait.
the robot, then hit an obstacle
I send the robot the commands to return to the den
the robot return to the den and ends.
```

Test Plan

To make the discussion easier, we introduce the following command abbreviations cril:

```
w : express the move {"robotmove":"moveForward", "time" :300}
s : express the move {"robotmove":"moveBackward", "time":300}
l : express the move {"robotmove":"turnLeft ",  "time ":300}
r : express the move {"robotmove":"turnRight",  "time ":300}
```

TESTPLAN :for each successful move, we add the move identifier to a **moves** string (initially empty).
We set the **time** value for the **w, s** moves to move the robot in 'small steps', so as to reach an **obstacle** with **N> 1** moves.
When the robot hits an obstacle, the string must take the following form:

```
moves:  "[lr]*[ws]+[wslr]*"        * : repeat 0 or more times
```

Then the robot returns to the den replacing the forward moves with the backward moves, the left moves with the right moves and vice versa. The return moves are saved in a second string. So the two strings must be the opposite of each other.

Project

ActorBasicJava

ActorBasicJava.java :
an abstract class that implements the concept of a message-driven entity that handles messages in FIFO order, by delegating the work to the abstract method **handleInput**.

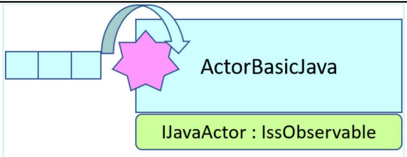
```
public abstract class ActorBasicJava extends Thread
   implements IJavaActor {...}
```

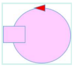
IJavaActor.kt :
defines the operations that each ActorBasicJava component must provide.

```
package it.unibo.interaction
interface IJavaActor : : IssActorObservable {
    fun myname() : String
    fun send(msg: String )
}
```

IssActorObservable.kt :
each ActorBasicJava component must be observable by other ActorBasicJava components

```
package it.unibo.interaction
interface IssActorObservable {
    fun registerActor(obs: IJavaActor)
    fun removeActor(obs: IJavaActor)
}
```



Actor symbol: 

1. Now, in our analysis and design phase, we can talk of two main types of components: **objects** (POJO) and **actors**.

2. An ActorBasicJava has the capability to interact with other actors (known by reference) by using the following local operations:

```
protected void forward(String msg, ActorBasicJava dest) { dest.send(msg); }
protected void request(String msg, ActorBasicJava dest) { dest.send(msg); }
protected void reply(String msg, ActorBasicJava dest) { dest.send(msg);   }
```

3. An ActorBasicJava can be considered a sort of 'local-microservice'

IssWsHttpJavaSupport

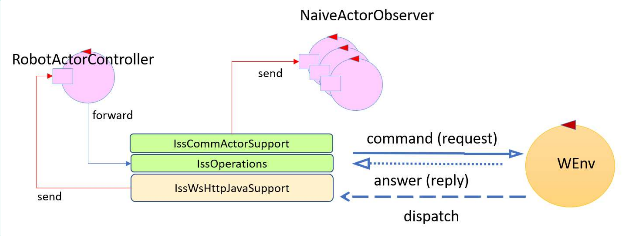
IssWsHttpJavaSupport.java:
a class that extends the WebSocketListener of the library OkHttp3. This class is able to send / receive messages via http/ws and is able to send information to ActorBasicJava.java actors registered as observers.

```
public class IssWsHttpJavaSupport extends WebSocketListener
    implements IssActorObservable, IssCommSupport, IssOperations {
```

IssCommActorSupport.kt:

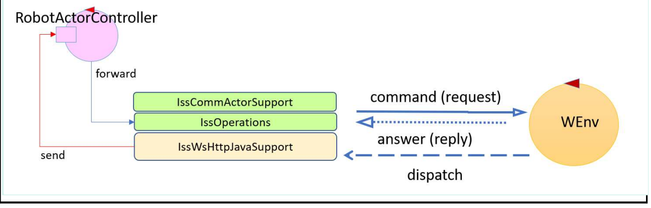
```
package it.unibo.interaction
import it.unibo.supports2021.ActorBasicJava
interface IssCommActorSupport  {
    fun isOpen() : Boolean
    fun close()
}
```

MainIssWsHttpJavaSupportUsage.java:
shows a demo of the usage of the IssWsHttpJavaSupport.java.



Example: the boundaryWalker (project it.unibo.virtualrobotclient)

MainRobotActorJava.java:
the boundaryWalker application as a system composed of ActorBasicJava.java components.



By Lucrezia Corsaro lucrezia.corsaro@studio.unibo.it - https://github.com/LucreziaCorsaro/CorsaroLucrezia-Iss