LABORATORIO DI INGEGNERIA DEI SISTEMI SOFTWARE

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 closed environment, delimited by walls that includes one or more devices (e.g. sonar) able to detect its presence;
- the robot has a **den** for refuge, located near a wall;
- the robot works as an explorer of the environment. Starting from its **den**, the robot moves (either randomly or - preferably - in a more organized way) with the aim to find the fixed obstacles around the **den**. The presence of mobile obstacles is (at the moment) excluded;
- since the robot is 'cautious', it returns immediately to the **den** as soon as it finds an obstacle. Optionally, it should also return to the den when a sonar detects its presence;
- the robot should remember the position of the obstacles found, by creating a sort of 'mental map' of the environment.

Requirement analysis

A closer confrontation with the custum has clarified that the customer intends:
NOUNS

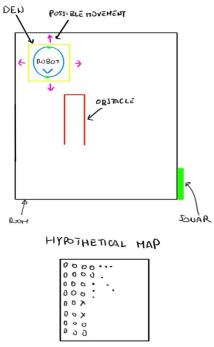
- "**robot**": a device able to execute move commands sent over the network, as described in the document VirtualRobot2021.html provided by the customer;
- "**closed environment**": a conventional (rectangular) room of an house, delimited by **walls**;
- "**sonar**": external device that is included in the boundary of the room that aquires information about the robot's position inside the room;
- "**den**": fixed starting position inside the room near a **wall** where the robot will be located before the software starts. It is also defined where the robot is initially oriented (North, South, East, West);

- "**obstacles**": **general objects** or even **walls** around or inside the boundary of the room. The obstacles can be fixed or mobile;
- "**mental map**": an abstract representation of the room where the robot is and the obstacles inside;

VERBS

- "**walk**": the robot moves freely in any direction. Its path can be random or organized;
- "**finds**": the robot finds an obstacle when the robot collides with it or if the robot's sonar detects its presence

Main user story

The robot is located in his starting **den** facing South. When the sofware application is activated the robot will randomly/in a controlled manner move inside the closed environment. Whenever it is found an **obstacle**, or the sonar detect its presence, the robot will have to return to its **den**. The application can be interrupted by the user and also in this case the robot must go back to his starting position. While the robot moves it aquires information about the environment it is in by creating a map. When the application terminates, the itinerary done by the robot must be that shown in the figure and a proper **TestPlan** should properly check this outcome.	

Problem analysis

We highlight that:

1. In the VirtualRobot2021.html: commands the customer states that the robot can receive move commands in two different ways:
   - by sending messages to the port 8090 using **HTTP POST**
   - by sending messages to the port 8091 using a **websocket**
2. With respect to the technological level, there are many libraries in many programming languages that support the required protcols.

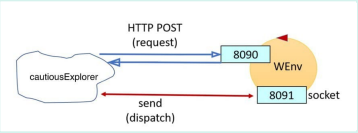
However, the problem does introduce an **abstraction gap at the conceptual level**, since **the required logical interaction** is always a **request-response**, **regardless of the technology** used to implement the interaction with the robot.

Logical architecture

We nust design and build a distributed system with two software macro-components:

1. the VirtualRobot, given by the customer
2. our **cautiousExplorer** application that interacts with the robot with a **request-response** pattern

A first scheme of the logical architecture of the systems can be defined as shown in the figure (for the meaning of the symbols, see the legenda)



It is observed that:

- To make out software application as much as possible independent from the undelying communication protocols, the designer could make references to proper **design** patterns such as **Adapter**, **Factory**, **Builder**, ext.
- It is quite easy to define **what the robot has to do** to meet the requirements:

  We can define a set of availlable movement directions for the robot (Down, Up, Left, Right). Then the robot is located to is **den** facing the Down direction. Until the robot doesn't encounter an obstacle or the user stops the application from running:

  1. send the robot random/organized movement commands for it to move;
  2. send the robot the **backwords commands** for it to go back to its starting position;
  3. check the final position if it is equal to the **den**;

Test plans

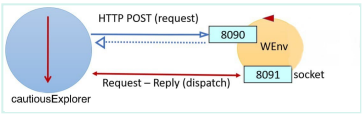
To check if the application fulfills the requirements, we could keep track of every moves done by the robot. Once a collision is detected, the sonar detects the robot's presence or the abort command is called we map backwards the moves into their opposites and we keep track of the movement made by the robot after the event and we compare the results.
For instance:

1. Let's define a **String moves="";**
2. Every time the robot moves we append the symbol corresponding to the direction taken (e.g. after the command **moveForward** we append a **"w"** and if a command moveLeft is sent after, we append a **"l"**. At the end **moves** will look like **"wl"**) ;
3. After a collision or an interruption (by sonar or user) is detected, we map **moves** to its opposit **String movesOpos="rb"** (we map backwards so **"wl"** becomes **"lw"** and we substitute each movement with its oposite **l -> r (right)**, **w -> b (back)**);
4. Then we clear **moves** and we repeat the point number 2;
5. At the end we compare the new value of **moves** with **movesOpos**;

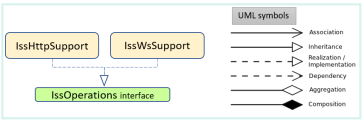
The last point can be easily checked with a TestUnit software.

Project

Nature of the application component

Our application "**cautiousExplorer**" will be a conventional Java program and it can be represended by a object with an internal thread, as is shown in the figure.	

Communication's abstraction layer

It is noticed from the technological detail from VirtualRobot2021.html that we have two different technologies for communicating with the robot. This gap can be resolved by adding an abstraction level creating a layered architecture, which is the simplest form of software architecture. A pattern usefull to fulfill the gap is the pattern Facade which allows the programmers to work through an object to minimize the dependencies on a subsistem (in our case the communication). So we will have an interface containing the basic method such as **forward**, **reply** and **request** which are common to both **HTTP** and **Websocket** protocols. The java interface defined in the project **boundaryWalk** named ***IssOperations.java*** will do the job. The way this interface can be used is shown in the image.	
It is possible to add another abstraction layer on the creation of the object that will handle the communication. We can use the pattern Factory embadded in a class that will initialize the protocol that will be used. The class ***IssCommsSupportFactory.java*** from **boundaryWalk** will do the job.	

Testing

Deployment

Maintenance

By Simone Mattioli email: simone.mattioli6@studio.unibo.it