

Introduction

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

Requirements

Design and build a software system that allow the robot described in [VirtualRobot2021.html](https://www.virtualrobot2021.html) to exhibit 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.

Delivery

The customer requires to receive the completion of the analysis (of the requirements and of the problem) by **Friday 12 March**. Hopefully, he/she expects to receive also (in the same document) some detail about the project.

The name of the file (in pdf) should be:

cognome_nome_ce.pdf

Requirement analysis

After interviewing client, menanig he associates whith nouns have been clarified:

- **closed environment**: similar to a room delimited by walls, the robot can't go out from this room.
- **den**: a place where the robot is initially positioned. Is not a fixed place, but can be changed by user discretion.
- **obstacles**: any object that cause a collision with if robot hits it. Their position are fixed since start of the work.
- **cautious explorer**: robot's behaviour is like a cautious explorer, if he hit any kind of obstacle, he come back to initial position that is **den**.
- **mental map**: the robot needs to remember all space that he have already explored and obstacles he find in his exploration.
- **(Optional) sonar**: is a system that acts on entire line where he is positioned and facing. In case the robot enter in sonar zone, he detect it and come back to den.

Regarding to actions (verbs):

- **moves**: robot can moves around the closed environment using all four principal directions.
- **find an obstacle**: when the robot collides with an obstacle.

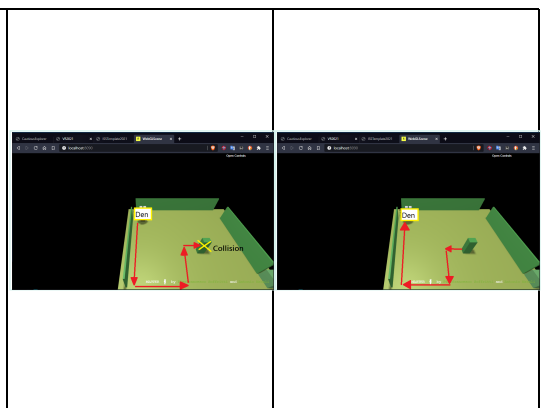
A first user story.

At the start of the work, den can be positioned near one of walls. Once work is started robot can moves in any position he want. All moves he performs are remembered so that is created a '**mental map**' of the environment in where he is. When, after a move, robot hit one of obstacles in environment, he registers its position in his environment 'mental map' and he comes back to the den.

User can't stop activity execution: system terminate independently after the completion of activity.

Application can be runned more time to build environment map. It is not necessary to complete map in one run.

In images is reported only an example of a possible path that robot can performs in one of his 'cautious' exploration.



Verification of expected results (Test plans)

It is necessary to verify that expected behavior for robot is respected, and at the end of execution positions of obstacles have been found.

Verification of correctness of robot's behavior and final environment map, it must be carried out by software, without an human user's interaction

Problem analysis

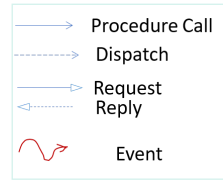
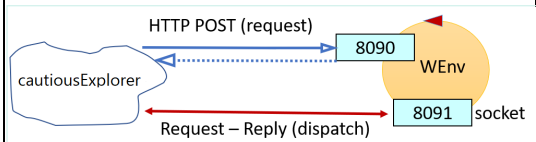
Aspects noted

1. It's about realizing a **distributed system** composed by two macro-component:
 - robot (virtual) provided by committent
 - Our application (**cautiousExplorer**) that send correct commands to robot so as to satisfy requirements
2. The robot can be moved by network in two different ways, like described in [VirtualRobot2021.html](#) **command** section:
 - Sending messages to port **8090** with HTTP POST protocol
 - sending messages to port **8091** with websocket technology
3. As described in requisites, it is strongly recommended to make a map (planimetry) of the environment in where the robot performs his exploration.
4. **Optionally** we have to handle the reactive component of the robot in case we want to implement sonar in our application.
5. As there are a lot of library in a multitude of programming languages that permit sending of these commands, it's not identified any significative **abstraction-gap** on operation plan.

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.

6. Estimate that a first application prototype, should be realized by Friday 12 March 2021 (at most).

Logical Architecture

<p>We do introduce the following terminology:</p> <ul style="list-style-type: none"> • Dispatch: a message '<i>fire and forget</i>': the sender does not expect any answer. • Request: the sender expects an answer sent using a Reply. • Invitation: the sender expects an ack. • Event: the sender '<i>emits information</i>' without specifying any receiver. 	
<p>We must design and build a distributed system with two software macro-components:</p> <ol style="list-style-type: none"> 1. the VirtualRobot, given by the customer 2. our cautiousExplorer application that interacts with the robot with a HTTP POST protocol or websocket technology. <ul style="list-style-type: none"> ◦ With HTTP POST protocol we have consequently a request-receive interaction, however in our case we don't need to handle the reply that robot send us. Considering that application doesn't necessarily have to handle reply sended us, we can build a system that use a dispatch pattern. In this way the execution continue without blocking until the end. ◦ Usind instead <i>websocket technology</i> it does not have this type of problem. <p>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)</p>	

There are many point on with we can do some observation:

- To make our **cautiousExplorer** software **as much as possible independent** from the underlying communication protocols, the designer could make reference to proper **design pattern**, e.g. **Adapter**, **Bridge**, **Facade**.
- We can say that our software is an application and not a database or a function or an object. In any case exact 'nature' of this software is left to designer.
- It's easy to describe what the robot has to do to meet requirements

Let us define **enum direction {UP,DOWN,LEFT,RIGHT}**. Those are moves that robot can do..

The robot start in the DEN position, that it could be different every time user run application.

- 1) send to the robot request to start exploration of room and continue to do it, until he collide with a wall or an o could be totally random or somehow handled.
- 2) After collision robot re-do in reverse same moves that he have performed previously, so that he return immediately
- 3) On every run of application robot 'remember' moves he do until a collision, so that he build a 'mental-map' of the

Test plans

To check that application fulfills requirements, it's necessary to keep track of the moves done by the robot until a collision and verify that robot follow same path to return to the den.

Two different ways to check the correctness have been devised:

1. TESTPLAN-IPOTESIS 1

Application at the start generate moves that robot has to follow during his exploration. Generation of these moves can be totally random or create following some criteria (maybe checking the 'mental map'). At this point the application record those moves in a string during robot exploration. After a collision he has to return to den following same moves but in reverse. To check the correctness, these moves are recorder in another string. At the end, these two strings are compared, and if they are exactly opposite, test is successfully passed.

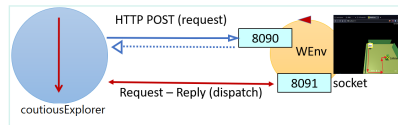
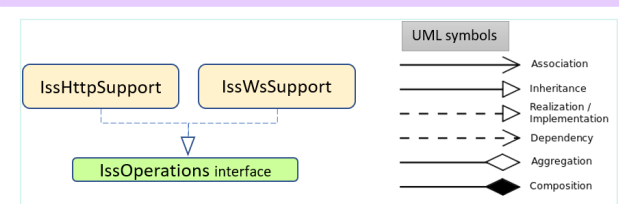
2. **TESTPLAN-IPOTESIS 2**: setting time for w,s moves, in order to obtain robot-unit movements, build incrementally a map of room after every move. In this way we could know position of robot after every command, and then, at the end of execution we could verify the path taken; for example (1 represent 'cells' taken by robot, X represent a collision 'cell')

```
|D, 0, 0, 0, 0,
|1, 0, 0, 0, 0,
|1, 0, 1, X, 0,
|1, 0, 1, 0, 0,
|1, 1, 1, 0, 0,
```

After a collision robot has to follow same path and return to den increasing by one cells on which it passes. At the end his position has to be the den and value of all trampled cells must be 2.

```
|D, 0, 0, 0, 0,
|2, 0, 0, 0, 0,
|2, 0, 2, X, 0,
|2, 0, 2, 0, 0,
|2, 2, 2, 0, 0,
```

Project

<p>Nature of the application component</p> <p>The cautiousExplorer application is a conventional Java program, represented in the figure as an object with an internal thread.</p>	
<p>A layered architecture: the basic communication layer</p> <p>To make the 'business code' as much independent as possible from the technological details of the interaction with the virtual robot (and with any other type of robot in the future), let us structure the code according to a conventional layered architecture, which is the simplest form of software architectural pattern, where the components are organized in <i>horizontal layers</i>.</p> <p>For each protocol we will introduce a proper support that implements the interface IssOperations.java</p> <pre>public interface IssOperations { void forward(String msg); void request(String msg); void reply(String msg); String requestSynch(String msg); }</pre> <p>These operations are introduced with reference to message-passing rather than to procedure-call. Thus, forward means just 'fire and forget', while request assumes that the called entity will execute a reply related to that request.</p> <p>requestSynch is introduced to facilitate the transition from procedure-call to message-passing.</p>	<p>The communication layers</p>  <p>The implementation of the IssHttpSupport.java is quite conventional, since the work is mainly done by the library org.apache.http.</p> <p>The implementation of the IssWsSupport.java is based on the library javax.websocket and requires a new 'style of programming' (that we will discuss later).</p>
<p>IssCommsSupportFactory</p> <p>The IssCommsSupportFactory.java provides a factory method to create the proper communication support by using a user-defined Java annotation related to the object given in input.</p>	<p>Using Java annotations</p> <p>The class IssAnnotationUtil.java provides utility methods to access the information specified in an annotation.</p>

<pre> IssCommsFactory public static IssOperations create(Object obj){ ProtocolInfo protocolInfo = IssAnnotationUtil.getProtocol(obj); ... } </pre>	
--	--

Configuration based on Java annotations (see [Lab ISS](#) | [About annotations](#))

Our user-defined annotation [IssProtocolSpec.java](#) allows us to configure the protocol by simply prefixing an application class with the annotation [@IssProtocolSpec](#).

```

@Target( value = {ElementType.CONSTRUCTOR,ElementType.METHOD, ElementType.TYPE} )
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface IssProtocolSpec {
    enum issProtocol {UDP,TCP,HTTP,MQTT,COAP,WS} ;
    issProtocol protocol() default issProtocol.TCP;
    String url() default "unknown";
    String configFile() default "IssProtocolConfig.txt";
}

```

Note that we do introduce also the possibility to specify the protocol configuration by using a file that includes sentences (in Prolog syntax) of the form:

```

spec( protocol( "WS" ), url( "localhost:8091" ) ).
spec( protocol( "HTTP" ), url( "http://localhost:8090/api/move" ) ).

```

The file-based configuration overcomes the configuration specified at code level, in order to augment flexibility at application level.

Example: using HTTP

```

@IssProtocolSpec(
    protocol = IssProtocolSpec.issProtocol.HTTP,
    url      = "http://localhost:8090/api/move"
)
public class UsageIssHttp {
    private IssOperations support;

    public UsageIssHttp(){
        support = IssCommsFactory.create( this
    }
    ...
}

```

The full code is in [UsagelssHttp.java](#).

Example: using WS

An example related to the usage of websockets can be found in [UsagelssWs.java](#). We will discuss it later.

The application component (from cril to aril)

Since we intend to make our 'business code' technology-independent also with respect to the robot, we introduce **a layer that makes the robot a 'logical entity'** able to 'talk' with clients in a **custom high-level language**, designed with reference to the application needs. In the following, we will name such a language as **aril** (**abstract robot interaction language**).

The abstract robot interaction language (aril)

The language that we will use to talk with our 'logical robot' is defined by the following grammar rule:

```
ARIL ::= w | s | l | r | h
```

Moreover, if we assume here that the 'logical robot' can be included in a circle of diameter of length **DR**, the meaning of the aril commands can be set as follows:

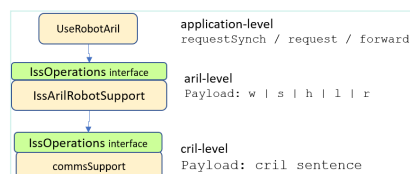
```

w : means 'go forward', so to cover a length equals to DR
s : means 'go backward', so to cover a length equals to DR
h : means 'stop moving'
l : means 'turn left of 90°'
r : means 'turn right of 90°'

```

From cril to aril ([IssArilRobotSupport](#))

The class [IssArilRobotSupport](#) is introduced as a component that translates **aril** commands into **cril** commands, thus working as an **adapter**.



The move-time of aril-commands is set by using the user-defined Java annotation: [RobotMoveTimeSpec](#)

```

@Target(value = { ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface RobotMoveTimeSpec {
    int ltime() default 300;
    int rtime() default 300;
    int wtime() default 600;
    int stime() default 600;
    int htime() default 100;
    String configFile() default "IssRobotConfig.txt";
}

```

The move-time can be also specified by means of a file that includes sentences (in Prolog syntax) of the form:

```
spec( htime( 100 ), ltime( 500 ), rtime( 500 ), wtime( 600 ), wstime( 600 ) ).
```

Testing

Deployment

Maintenance

My git repo: https://github.com/LeoManto/Mantovani_Leonardo

By Mantovani Leonardo email: leonardo.mantovani2@studio.unibo.it

