Lab ISS | the project resumableBoundaryWalker

Introduction

This case-study starts to deal with the design and development of proactive/reactive software systems which work under user-control.

Requirements

Design and build a software system (named from now on 'the application') that leads the robot described in *VirtualRobot2021.html* to walk along the boundary of a empty, rectangular room under user control.
More specifically, the **user story** can be summarized as follows:

the robot is initially located at the **HOME** position (i.e., top-left corner).
the application presents to the user a **consoleGui** (i.e., GUI with 2 buttons: STOP and RESUME).
when the user hits the button **RESUME** the robot **starts or continue to walk** along the boundary, while updating a **robot-moves history**;
when the user hits the button **STOP** the robot stops its journey, waiting for another **RESUME** ;
when the robot reachs its **HOME** again, the application *shows the robot-moves history* on the standard output device.

Requirement analysis

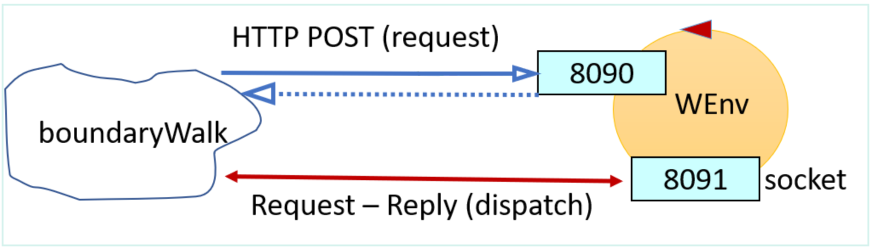
- for room: a conventional (rectangular) room of an house;
- for boundary: the perimeter of the room, that is physically delimited by solid walls;
- for robot: a device able to execute move commands sent over the network, as described in the document VirtualRobot2021.html provided by the customer;
- for walk: the robot moves forward, close to the room walls;
- for robot-moves history: the application driving the robot must memorize somewhere the moves of the robot in order to print them when needed.

Problem analysis

We highlight that:
- 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

Logical architecture

We nust design and build a distributed system with two software macro-components:
- the VirtualRobot, given by the customer
- our boundaryWalk application that interacts with the robot



We must observe that the requirements imply a reactive behaviour from the robot. The general commands sent to the robot can be sent through the usual **Fire and forget** pattern. The information created by the user clicking on STOP/RESUME is an **event**.
The WebSocket protocol is the bast candidate for handling this fire and forget communication pattern. Also, it is lighter than the HTTP protocol.

The designer(s) should design an handler(s) component to handle the reactive behaviour that the robot must exhibit accordingly to requirements.

We observe that:

- The specification of the exact 'nature' of our boundaryWalk software is left to the designer. However, we can say here that is it not a database, or a function or an object.
- To make our boundaryWalk software as much as possibile independent from the undelying communication protocols, the designer could make reference to proper design pattern, e.g. Adapter, Bridge, Facade.

With respect to the technological level, there are many libraries in many programming languages that support the required protcols. The following resources could be usefully exploited to reduce the development time of a first prototype of the application:

1. The Consolegui.java (in project it.unibo.virtualrobotclient)
2. The RobotMovesInfo.java (in project it.unibo.virtualrobotclient)
3. The RobotInputController.java (in project it.unibo.virtualrobotclient)
4. The mapUtil.kt to build and then show the map that has been discovered by the robot.

... TODO

individua e propone (motivandola) una lista di priorità per il soddisfacimento dei requisiti

The expected time required for the development of the application is (no more than) 6 hours.

Test plans

presenta ogni TestPlan collegandolo ad una user story e in modo 'concreto' (non come un insieme di intenti)
propone diverse tipologie di scene/situazioni per il testing
Test plan to check that the robot does correctly the "boundary walk".

```
let us define String moves="";
for 4 times:
        1) send to the robot the request to execute the command moveForward;
        if the answer is 'true' append the symbol "w" to moves and continue to do 1);
        2) when the answer of the request becomes 'false',
        send to the robot the request to execute the command turnLeft and append the
        symbol "l" to moves
```

Test plan for the robot to start after the RESUME button is hit.

```
precondition: the robot is idle.
then:
        1. the handler receives the RESUME event
        2. the component(s) controlling the robot do:
                for 4 times:
                        1) send to the robot the request to execute the command moveForward;
                        if the answer is 'true' append the symbol "w" to moves and continue to do 1);
                        2) when the answer of the request becomes 'false',
                        send to the robot the request to execute the command turnLeft and append
                        the symbol "l" to moves
```

Test plan for the robot to check that the robot stops -- after a STOP hit -- and wait the RESUME button to be hit.

```
precondition: the robot is moving. If the robot is idle we do nothing.
then:
        1. the handler receives the STOP event
        2. the component(s) in control of the robot do:
                1. send a halt command to the robot.

NOTE: It may happen that the robot receives a stop when, e.g., it is doing the
third iteration. When it resumes it must start with the beginning of the third
iteration, hence the robot must save somewhere the iteration no. before stopping.

NOTE: if the test about the RESUME event does not fail and memorizes properly its
status about iteration number, then we have covered all the requirements with
these tests.
```

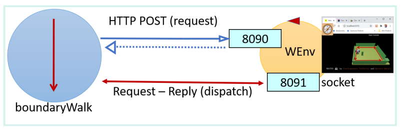
Project

presenta dettagli di progetto che permettono una precisa implementazione da parte di chi legge
pone in evidenza le parti di funzionamento proattivo e quelle di funzionamento reattivo

Nature of the application component

The ***boundaryWalk*** application is a **conventional Java program**, represented in the figure as an object with internal threads.

Project Architecture



A layered architecture: the basic communication layer

To make the 'business code' as much independent as possibile 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 *horizontal layers*.

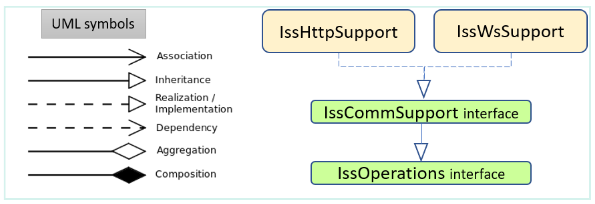
For each protocol we will introduce a proper support that implements the interface ***IssOperations.java***

```
public interface IssOperations {
    void forward( String msg ) ;
    void request( String msg );
    void reply( String msg );
    String requestSynch( String msg );
}
```

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.

requestSynch is introduced to facilitate the transition from procedure-call tomessage-passing.

The communication layers



The implementation of the ***IssHttpSupport.java*** is quite conventional, since the work is mainly done by the library **org.apache.http**.

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).

IssCommsSupportFactory

Using Java annotations

The *IssCommsSupportFactory.java* provides a factory method to **create** the proper communicartion support by using a **user-defined Java annotation** related to the object given in input.	The class *IssAnnotationUtil.java* provides utility methods to access the information specified in an annotation.

The application component

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 out 'logical robot' is defined by the following grammar rule:

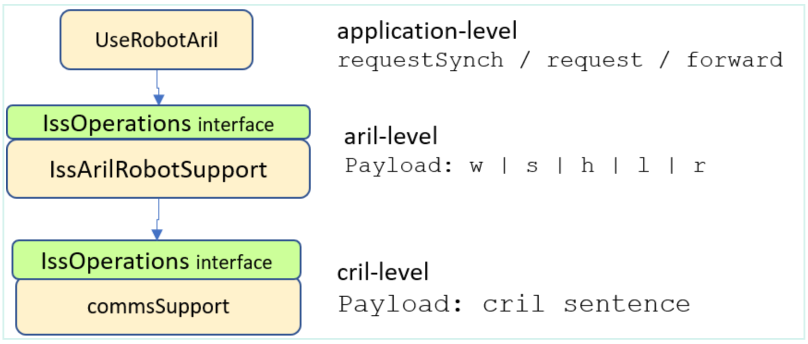
```
ARIL ::= w | s | l | r | h
```

Morover, if we assume here that the 'logical robot' can be included in a circle of diamater 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

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 ) ).
```

The usage of a websocket library **breaks** the unique flow of control of the application into several threads. In order to exploit in a structured way the asynchronicity of the interaction, the reference design pattern is the **Observer pattern**.

To facilitate the work of the application designer, we will introduce new resources, including the idea of a *support for (high-level) communications that provides operation to add/remove observers*, by implementing a proper interface.

The IssObserver interface

At the moment, this interface defines a method handleInfo that accepts two types of arguments

```
public interface IssObserver {
    public void handleInfo(String info);
    public void handleInfo(JSONObject info);
}
```

Our **observable supports** must implement an interface that adds new operations to our high-level communication interface:

The IssCommSupport interface

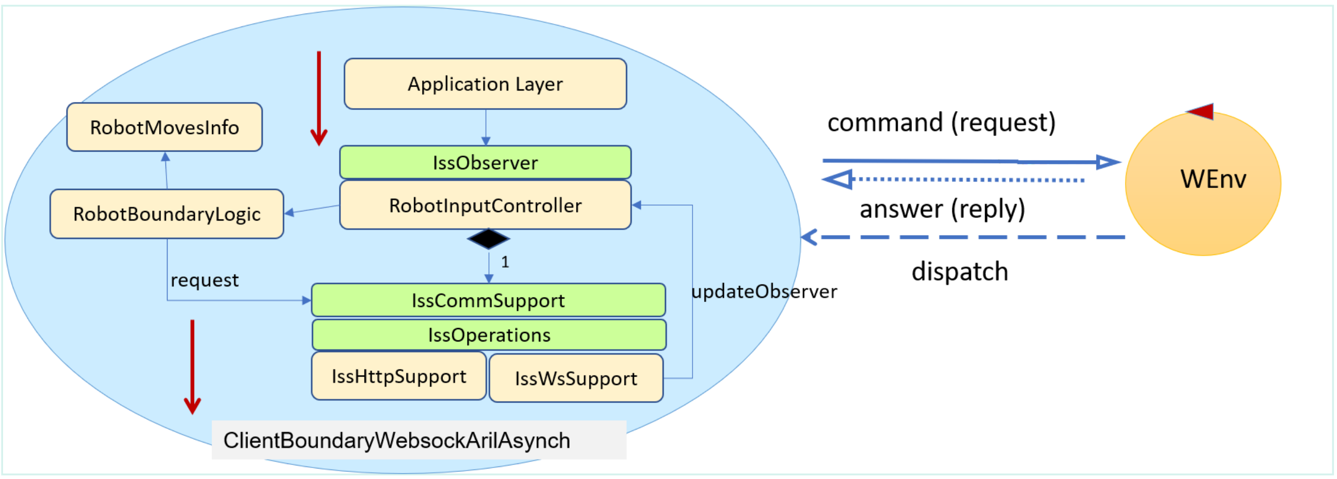
```
public interface IssCommSupport extends IssOperations {
    void registerObserver( IssObserver obs );
    void removeObserver( IssObserver obs );
    void close();
}
```

Zooming into the application

Class annotated with ArilRobotSpec,
Key point: Proactive and Reactive behaviour and **Single Responsibility Principle**.

- The business logic is defined in an object of class **RobotBoundaryLogic** that is called by the observer **RobotInputController** initialized to use the **aril** command-move language.
- In this way, we have further removed any detail related to the interaction with WEnv from the **RobotBoundaryLogic**.
- Moreover, the details related to the construction of the robot-moves history are embedded in the class **RobotMovesInfo**..

Zoomimg into the application



Requires 4 Threads, because of the IssWsSupport.

Since we have to stop/resume the robot when it receives the STOP/RESUME command by the user via the GUI, we use the class Consolegui. This imply that our RobotInputController becomes a reactive (to the command received by the GUI) component.

Testing

Deployment

The deployment consists in the commit of the application on a project named **iss2021_resumablebw** of the MY GIT repository (**RRR**).

The final commit commit has done after **XXX** hours of work.

Maintenance

By Andrea Zanni email: andrea.zanni8@studio.unibo.it