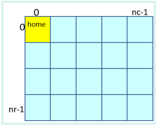
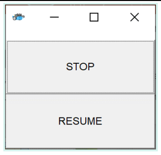
Lab ISS | the project resumableBoundaryWalker

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, as shown in the picture on the rigth	
the application presents to the user a **consoleGui** similar to that shown in the picture on the rigth	
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 stop 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.	

Delivery

The customer **hopes to receive** a working prototype (written in Java) of the application by **Monady 22 March**. The name of this file (in pdf) should be:

```
cognome_nome_resumablebw.pdf
```

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;
- "**room**": a conventional, empty and rectangular room of an house, delimited by **walls**;
- "**home position**": fixed starting position inside the room near a **wall** where the robot will be located before the software starts. The costumer initialized the **home** in the up left corner, facing South;
- "**robot-moves history**": the movement performed by the robot during his jurney. This has to be shown when the robot complete the entire boundary and is back to its home position;
- "**consoleGui**": a user interacting component that allow the costumer to send fixed commands to the robot;

VERBS

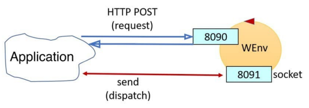
- "**walk**": the robot moves freely in any direction. Its path can be random or (**preferably**) organized. This constitutes also the **pro-active** part of pur system;
- "**resume**": starts the robot to let him finish the walk along the boundary;
- "**stop**": stops the robot in the position it is currently at allowing the user to resume it later;

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 protocols.
   - 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.
3. Two types of control language are made availlable from the costumer: the **cril** and the **aril**;
   - cril: we need to specify the time lenght of the command when we want to send a message;
   - aril: the time lenght of the command is fixed;
4. The aril language is prefered at test time because we can already know the desired results. For the same reason a controlled way of moving the robot is prefered.
5. A Gui is needed for the user to send commands to the robot. The calss Consolegui.java can be usefull to fulfil this task;
6. By adding a Gui component we define a view component that can interact with the robot throw a MVC pattern;

Logical architecture

We nust design and build a distributed system with two software macro-components: 1. the VirtualRobot, given by the customer 2. our **application** which 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)	 HTTP POST (request) 8090 WEnv Application 8091 socket send (dispatch)

Test plans

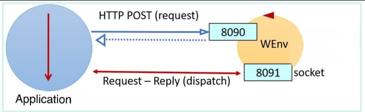
To check if the application fulfills the requirements, we could keep track of every moves done by the robot.

1. 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. when the robot has completed the room's boundary we compare the value of **moves** with the **expected result**;

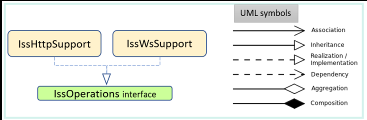
The last point can be easily checked with a TestUnit software.

Project

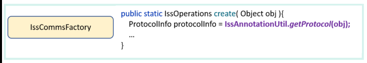
Nature of the application component

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.	 HTTP POST (request) 8090 WEnv 8091 socket Request – Reply (dispatch) Application

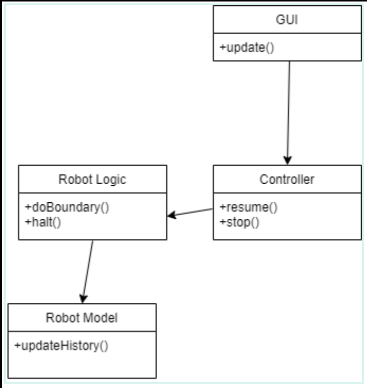
Communication's abstraction layer

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**,	 IssHttpSupport IssWsSupport IssOperations interface UML symbols Association Inheritance Realization / Implementation Dependency Aggregation Composition

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.	 IssCommsFactory
	`public static IssOperations create( Object obj ){`
	`    ProtocolInfo protocolInfo = IssAnnotationUtil.getProtocol(obj);`
	`    ...`
	`}`

Pattern MVC for GUI

The application can be much easilly managed by implementing a MVC pattern. This way we isolate the GUI and the robot behavior making them communicate through a controller that manipulate the businness logic of the robot. A first architecture can be seen in the picture on the right;	 GUI
	+update()
	Robot Logic
	+doBoundary()
	+halt()
	Controller
	+resume()
	+stop()
	Robot Model
	+updateHistory()

Workflow - Priority list

We define a list of priority for the fulfillment of requirements. The smallest the list index the higher the priority:

1. Create a stable and efficient communication system that can handle both HTTP and websocket. Otherwise if we are not apble to communicate with the robot there is no point of doing everything else;
2. Add to the communication system the message format (cril, aril). A great solution would to make the user able of using both languages;
3. Make the robot move around the boundary;
4. Create a Gui that can allow the user to make the robot do the boundary;
5. Create another button that allow the user to stop the robot;
6. Use the information received from the robot to create the moves-history;

Testing

Deployment

The deployment can be found in the project named **iss2021_resumablebw** inside my git repository.

The final commit commit has done after **5** hours of work.

Maintenance

By studentName email: simone.mattioli6@studio.unibo.it