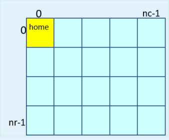
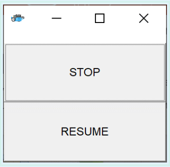
# Lab ISS | the project resumableBoundaryWalker (Landolfi Luca)

## 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

- **room**: a conventional rectangular room, as found in all buildings, with no obstacle ("empty")
- **boundary**: perimeter of the room, physically bounded by solid walls
- **robot**: a device capable of moving by receiving commands via the network, described in **VirtualRobot2021.html**.
- **under user control**: the application moving the robot can be paused and resumed by the user from a **consoleGui**. As stated in the user story above
- **Home**: starting position of the robot placed in the top left corner
- **consoleGui**: a user interface shown to the user composed of two button: one to stop and one to resume the application

For the verbs:

- **walk**: the robot must move forward, hugging the walls of the room.
- **starts or continue to walk**: the robot application will be started by the **consoleGui**
- **while updating a robot-moves history**: **?????????** the application have to mantain a robot move history
- **robot stop its journey**: robot finish his move and pause until he recive a resume command
- **robot reachs its HOME again**: the robot complete the boundary retourning in the **HOME** position
- **application shows**: print the output on the terminal
- **output device**: the pc hosting the application

### User Stories

User story were just defined by the customer in the requirements.
We will make reference to them for the rest of the project.

## Problem analysis

### Revelant aspects

In the VirtualRobot2021.html: commands the customer states:

1. 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. that the robot can send information:
    - Response to move request (**endmove**) replying using: **HTTP POST** and also **websocket** messages
    - Sonar information or collision detections from sensor by sending message using a websocket from the port 8091
3. 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** are **asynchronous**. Application need to handle asynchronous messages coming from the **robot** at any time of the execution. Wenv can ***dispatch*** some information at any time of the execution, even while the application is executing commands (***request-response***) on the robot.

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)

Also resources from the VirtualRobotClient can be useful:

- The Supports Package
- The Annonation Support Package
- The RobotBoundaryLogic.java (in project it.unibo.virtualrobotclient)

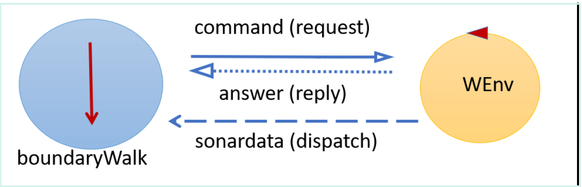
### Logical Architecture

| | |
|---|---|
| A **distributed system** with two software macro-components will | |

match the requirements:

1. the **VirtualRobot**, given by the customer
2. our **Application** interacts with the robot with asynchronus communications

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)



We observe that:

- The specification of the exact 'nature' of our ***cautiosExplorer*** software is left to the designer. However, we can say here that is it **not a database, or a function or an object**. And to properly handle the **asynchronus communication** on the webscoket the designer could make reference to the **Observer** design pattern.
- The designer can use a MVC architectural pattern to define the architectural layer of the application to a more organized and structured project.

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

## Test plans

To check that the application fulfills the requirements, we could keep track of the moves done by the robot. For example:

```
...
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
```

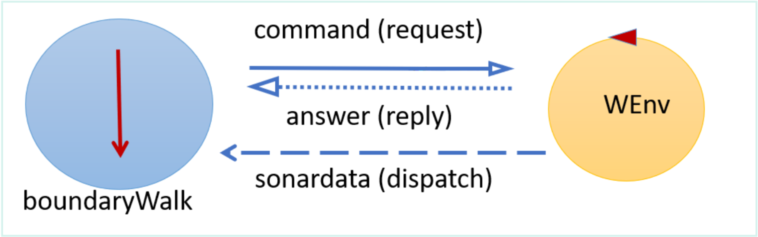
In this way, when the application terminates, the string **moves** should have the typical structure of a ***regular expression***, that can be easily checked with a TestUnit software:

```
moves:  "w*lw*lw*lw*l"      * : repetion N times(N>=0)
```

## Project

### Nature of the application component

The application is a **program**, represented in the figure as an object with an internal thread.



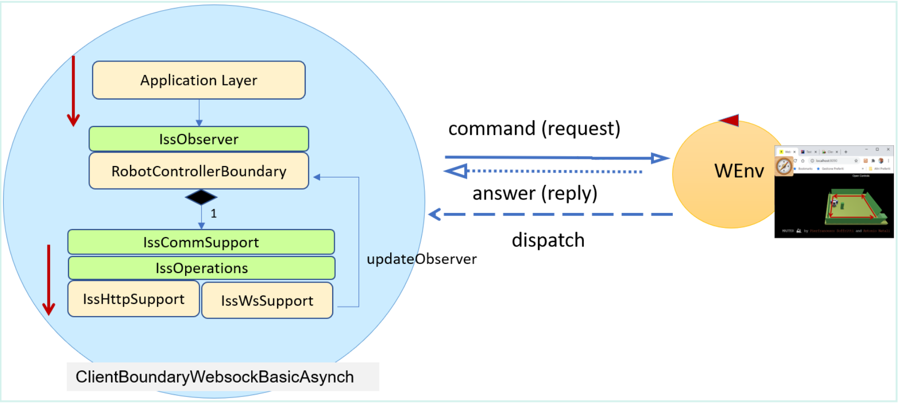
### A layered architecture

To make the 'business code' as much independent as possibile from the technological details of the interaction with the virtual robot

**Zoomimg into the application**

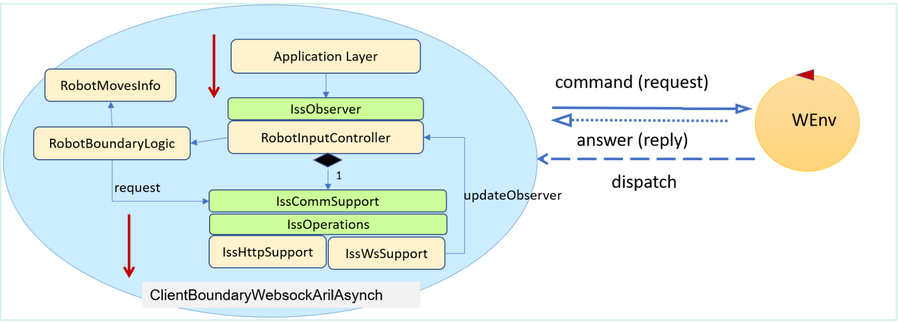
(and with any other type of robot in the future), let us structure the code according to a conventional ***layered architecture***, which is the simplest form of software architectural pattern, where the components are organized in *horizontal layers*.

The architecture of this project make refernce to the **BoundaryWalk Project** and the **VirtualRobot Project**. The first layered structure is reported in the **BoundaryWalk**. the **observer pattern** is also used to manage asynchronous communication, as explained in **VirtualRobotClient** document. We can see the architecture in the image on the left.



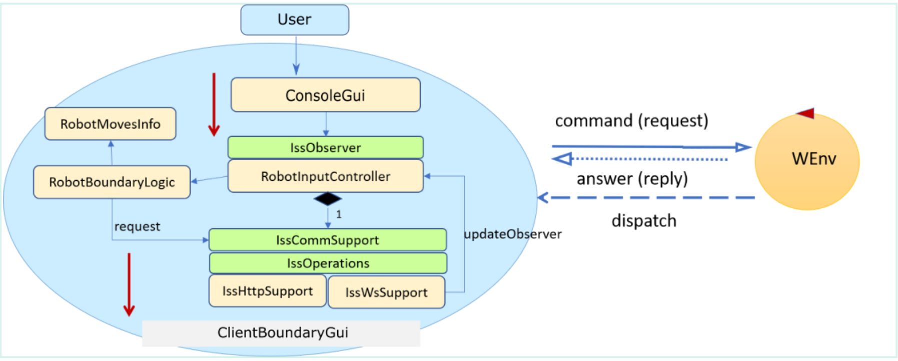
## Adding isolated business logic to save the robots moves history

Referencing **VirtualRobotClient** we isolate the business logic to mantain the robot moves info as in the image, adding two components **RobotBoundaryLogic** and **RobotMovesInfo**.



## User Interface components: ConsoleGui

The user interface have to start, stop and resume the Robot. To do this, we add the Consolegui.java as an observer observing the robot controller **RobotInputController**. This observer will be updated by the User Interface (Gui) and will send the action from the gui to the **RobotInputController** that will handle them and stop or resume the robot. In this way we do not have to change any component functionality.



The imported class that will be modified are:

- the **Controller** class (RobotInputController.java) to handle messages coming from the **ConsoleGui**
- the **BoundaryLogic** class (
- The RobotBoundaryLogic.java) to modify the boundaryInit method making the robot starts with a resume command and restart wen it finishes the boundary

Aslo we need to add a min class **ClientBoundaryGui** (ClientBoundaryGui.java) to instantiate the Gui, supports and the controller.

## Testing

## Deployment

The deployment consists in the commit of the application on a project named **lan.luca.iss2021_resumablebw** of the MY GIT repository ( **https://github.com/LucaLand/LandolfiLuca** ).

The final commit commit has done after **3** hours of work.

## Maintenance

By LUCA LANDOLFI email: luca.landolfi3@studio.unibo.it
GitHub : https://github.com/LucaLand/LandolfiLuca