

Lab ISS | the project resumableBoundaryWalker


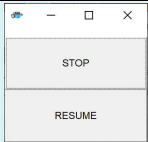
Introduction

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 right	
the application presents to the user a consoleGui similar to that shown in the picture on the right	
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 reaches its HOME again, the application <i>shows the robot-moves history</i> on the standard output device.	

Delivery

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

cognome_nome_resumablebw.pdf

Requirement analysis

- 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 **room**: a conventional (rectangular) room of an house;
- for **boundary**: the perimeter of the room, that is physically delimited by solid **walls**;

- for **home**: the corner in the left upper part of the room from which the robot starts moving;
- for **journey**: the path made by the robot around the room;
- for **user control**: user can stop or resume the journey of the robot;
- for **robot-moves history**: a representation of the moves made by the robot;
- for **consoleGui**: a graphical user interface which can be used from the user in order to resume/stop the journey

The customer does not impose any requirement on the programming language used to develop the application.

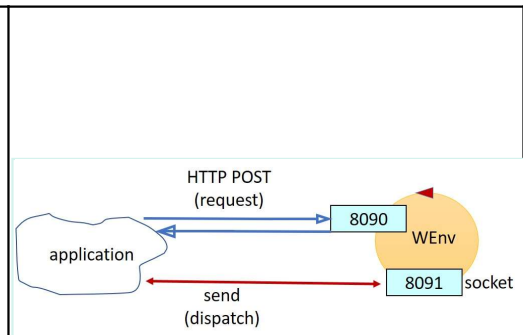
Problem analysis

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

1. the **VirtualRobot**, given by the customer
2. our **application** that interacts with the robot

A first scheme of the logical architecture of the systems can be defined as shown in the figure. Our application should

- send a **request** to WEnv for the execution of a robot-move command or for the resume/stop commands
- handle the information sent by WEnv to application as a **dispatch** carrying the reply sent by WEnv to the robot-move command request



Robot-Moves History

The business logic is able to build two different types of robot-moves history:

- a string that represents the robot path expressed as a sequence of moves. For example:

```
wwwlwwwlwwwlwwwl
```

- the places that the robot has explored, represented within a map of the room. For example

```
|r, 1, 1, 1, 1,
  |1, 0, 0, 0, 1,
  |1, 0, 0, 0, 1,
  |1, 1, 1, 1, 1,
```

In this representation, we suppose that:

1. r means: cell occupied by the robot
2. 0 means: cell not explored
3. 1 means: cell explored
4. X means: cell occupied by an obstacle

We observe that:

- The specification of the exact 'nature' of our **application** software is left to the designer. However, we can say here that is it **not a database, or a function or an object**.

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:

```
the robot starts from the HOME position, facing south (DOWN)
when the RESUME button is pressed:
    the robot starts to walk forward in small steps
    the application saves the moves
when the robot hits a wall:
    the robot turn left
    the application saves the moves
If at any time the STOP button is pressed:
    the robot stops its movement
    wait the action on resume button
the robot returns to the HOME position and show the 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:

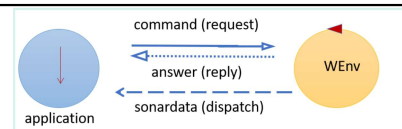
```
moves: "w*lw*lw*lw*" * : repeton N times(N>=0)
```

Project

The software system is focused on asynchronous interaction between the application and the robot, so it will be used WebSocket.

Nature of the application component

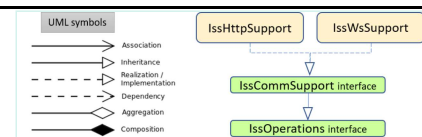
The application is a conventional Java program, represented in the figure as an object with internal threads.



A layered architecture

There are two ways of communication between the two macro-components, but the code must be as much independent as possible from the chosen one. This problem can be solved with a layered structure. For exemple, the pattern Facade can be used to hide the complexities of the larger system and provide a simpler interface shared by the two ways of communication. This interface can be structured with four basic methods, which are common to both HTTP and Websocket protocols. This methods can be:

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



The method forward sends a message without waiting for the reply. On the contrary the method request waits for the answer sent by the method reply. Instead, the method requestSynch is introduced to facilitate the transition from procedure-call to message-passing.

Observable supports

In order to exploit in a structured way the asynchronicity of the interaction, the reference design pattern is the Observer pattern. This pattern introduces an interface `IssObserver` that defines a method that accepts two type of arguments:

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

This method handle the info related to the movement of the robot.

This observable supports must implement an interface that adds new operations to the high-level communication interface:

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

This interface manages the observers.

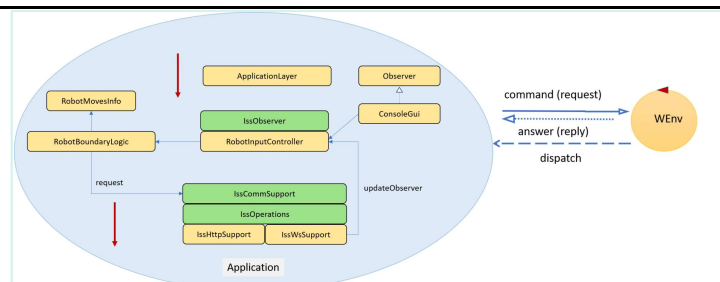
The figure shows an exemple of communication layer with observable support.

Creation method

A Factory can be used to create the chosen communication support. This pattern provides an interface to create an object, but lets the subclasses decide which object to instantiate. The Factory incorporates the creation of the necessary objects to handle the communication with the chosen protocol. This Factory must provide factory methods that return an object of type `IssCommSupport`.

Business logic

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. The details related to the construction of the robot-moves history are embedded in the class `RobotMovesInfo`. `ConsoleGui` implements `Observer`, when the resume/stop buttons are pressed, the update method interacts



with RobotInputController in order to complete the required operation

The figure shows an exemple of application.

Testing

Deployment

The deployment consists in the commit of the application on a project named **iss2021_resumablebw** of the MY GIT repository (<https://github.com/dbattaglia97/battagliadario>).

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

Maintenance

By Dario Battaglia email: dario.battaglia2@studio.unibo.it

