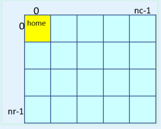
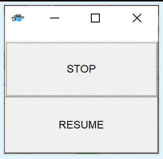
Lab ISS | the project resumableBoundaryWalker

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

The constumer requires to design and buid a software system that is able to drive a *robot* to walk along the boundary of an empty, rectangula room under user control. The interview with the constumer clarified the meaning of the used nouns that are the following:

- **robot**: device already owned by the consumer and able to walk by receiving commands. The robot is fully described in VirtualRobot2021.html, a document provided by the consumer;

- **room**: a conventional room found in all ordinary buildings;
- **boudary**: the perimeter of the physical room delimited by **walls**;
- **home**: the start point of the robot, located at the top left corner of the rectangular room;
- **consoleGui**: the graphical user interface that the application must present to the user;
- **button**: a button into the consoleGui that when pressed produces a certain action; the consumer requested two button:
  - **RESUME**: when the user press this button, the robot start or continue to walk around the boundary;
  - **STOP**: when the user press this button, the robot stop to walk waiting that the user press the resume button;
- **robot-moves history**: sequence of the movements performed by the robot.

About the meaning of the verbs used in the interaction with the consumer:

- **walk**: the ability of the robot to move thanks to the means of locomotion it is equipped with; constumer also specifies that the robot must move forward flanking the walls.

The robot is able to receive command in *cril language* coded into JSON strings that can be **encapsulated in HTTP POST messages** or **sent on a websocket** as specified in VirtualRobot2021.html. Consumer also required that **the working prototype must be written in** ***Java***.

Problem analysis

Relevant aspects

1. From the requirement analysis it emerged that the system that is about to be implemented must communicate with the robot via HTTP or via websocket using commands coded in JSON String. In order to do this it need to create a **distributed system** with two main components:
   - the **robot** provided by the consumer;
   - an **application** able to send the command to the robot.

   The application will be called ***resumableBoudaryWalker***.

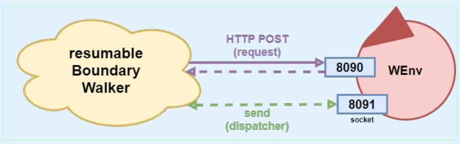
2. As anticipated, the robot and the application can communicate:
   - sending messages to the port 8090 with **HTTP Request/Response**;
   - sending messages to the port 8091 using **websocket**.

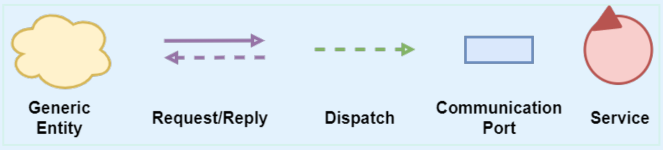
   In addition to, as specified, the commands must be coded in **JSON string**.

To realize communication the application must have a software part able to use HTTP or websocket and another part able to parse and use JSON data. There are lots of library for almost all languages but **there is an operative abstracion-gap**, since **the required logical interaction is not always a request-response**. In particular, to realize the asynchronous interaction with HTTP are needed little code tricks.

Logical Architecture

In the relevant aspects it was said that system has two main components then logical architecture is the following:

 resumable Boundary Walker HTTP POST (request) 8090 WEnv 8091 socket send (dispatcher)	For now **resumableBoundaryWalker** it's a generic component (entity) and it will be clarified in progect phase. **WEnv** is the environment of the robot able to receive commands via HTTP or websocket. About interaction instead: • the use of **HTTP** seems very suitable and very portable thanks to the existence of lots of API; • the use of **websocket** could be much more efficent decreasing the overhead costs of HTTP messages and in addition to the usage of this component allows asynchronous interation in a natural way.



We observe that:

- The specification of the exact 'nature' of our *resumableBoundaryWalker* 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 *resumableBoundaryWalker* 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**. **Observer**.

Reasources already available

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)

Application Logic

It is quite to define **what the robot has to do** to meet the requirements:

```
let us define emum direction {UP,DOWN,LEFT,RIGHT}
let us define list robot-history
the robot start in the HOME position, direction=DOWN
the gui present to the user the buttons RESET and STOP

when user press RESUME:
  for 4 times:
        1) send to the robot the request to execute the command moveForward
           and continue to do it updating the robot-history list,
           until the answer of the request becomes 'false'
           or the user press STOP button
        2) if the user pressed STOP, wait until RESUME is pressed;
        3) send to the robot the request to execute the turnLeft
  print robot-history contents to stdOut
```

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

Test plans

To check that the application requirement for doing boundary, 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
```

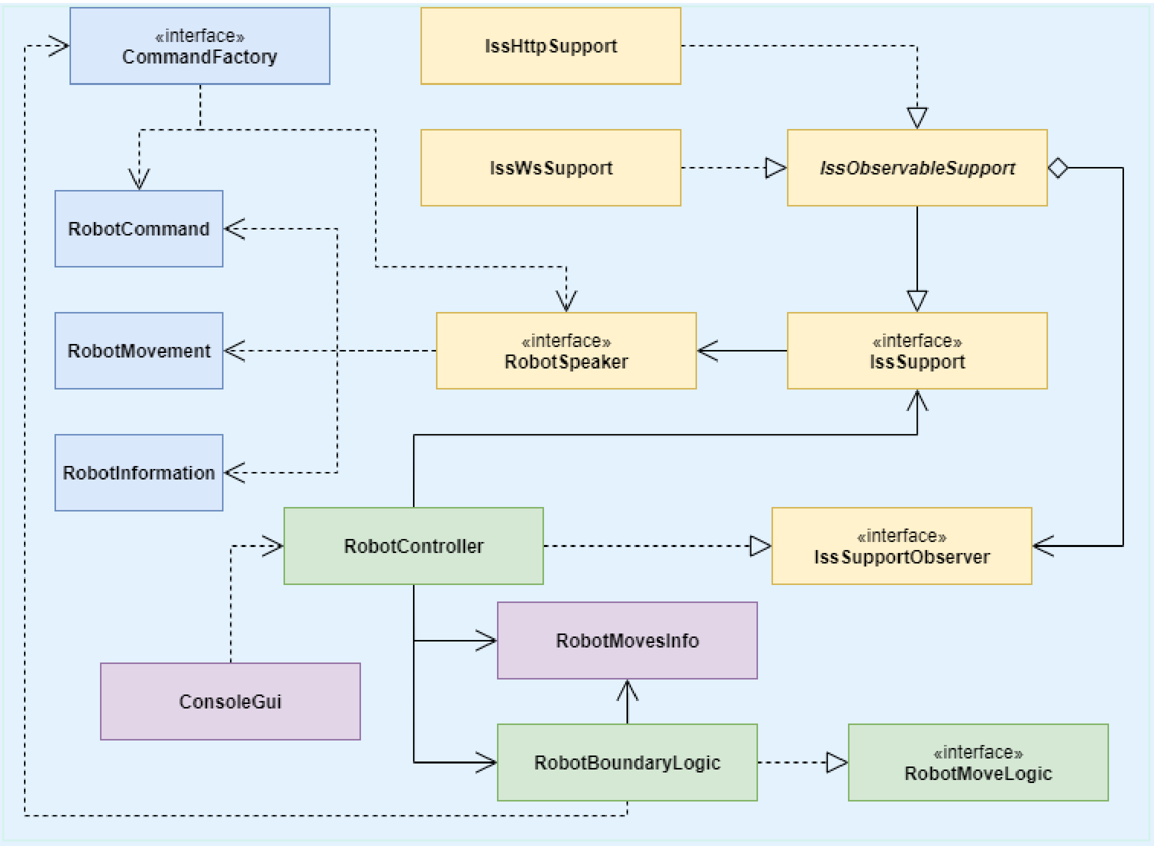
In this way, when the application terminates, the string **moves** should have the typical structure of a *regular expression*, that can be easily checked with a TestUnit software:

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

Project

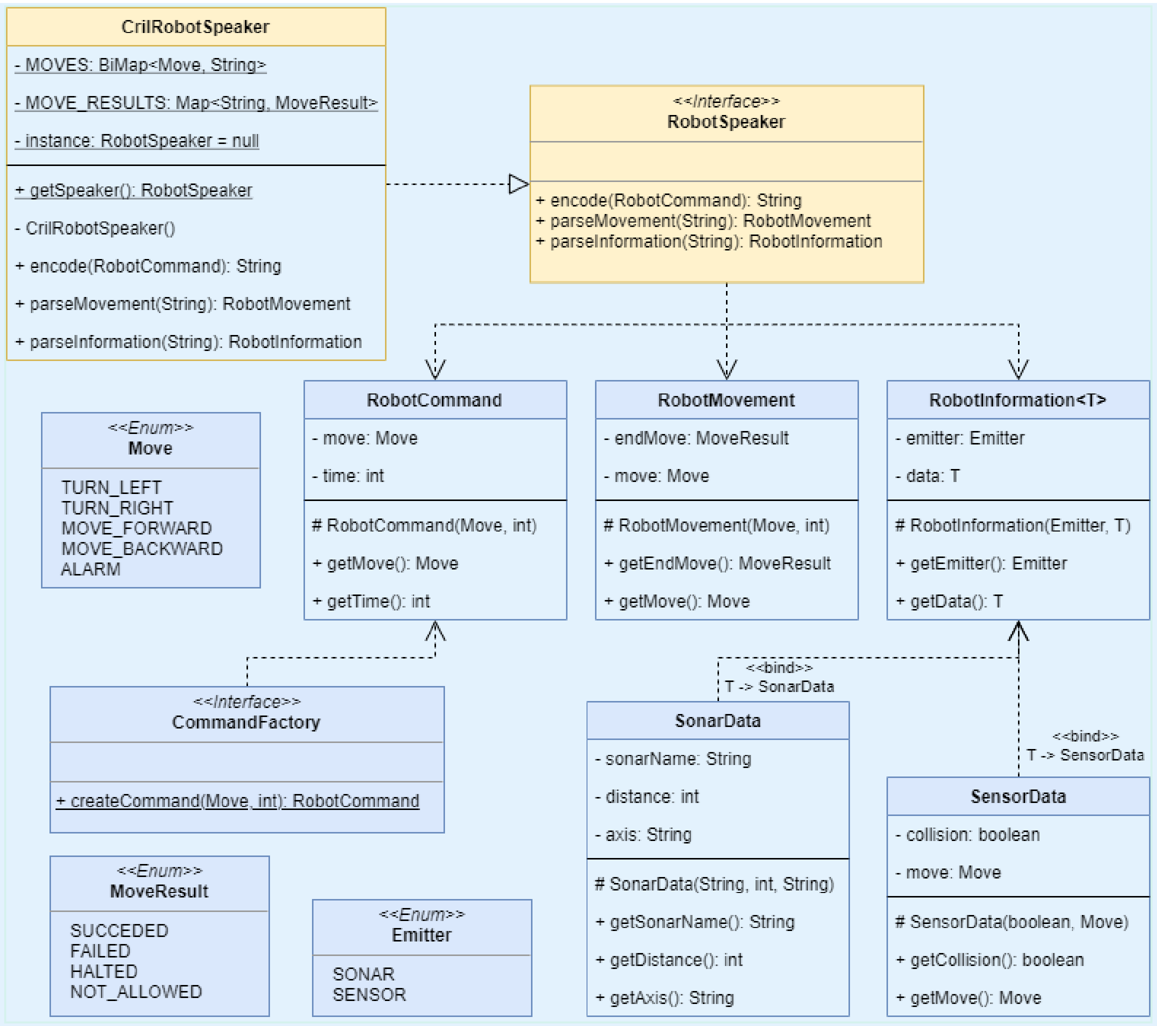
Nature of the application component

The ***resumableBoundaryWalker*** application is a **conventional Java program**, represented by multiple levels. The entire application can be represented by the following UML that will be zommed soon:



Command Level (package **it.unibo.resumableBoundaryWalker.commands**)

This package provides the necessary classes to map the messages sent via network in the object world.



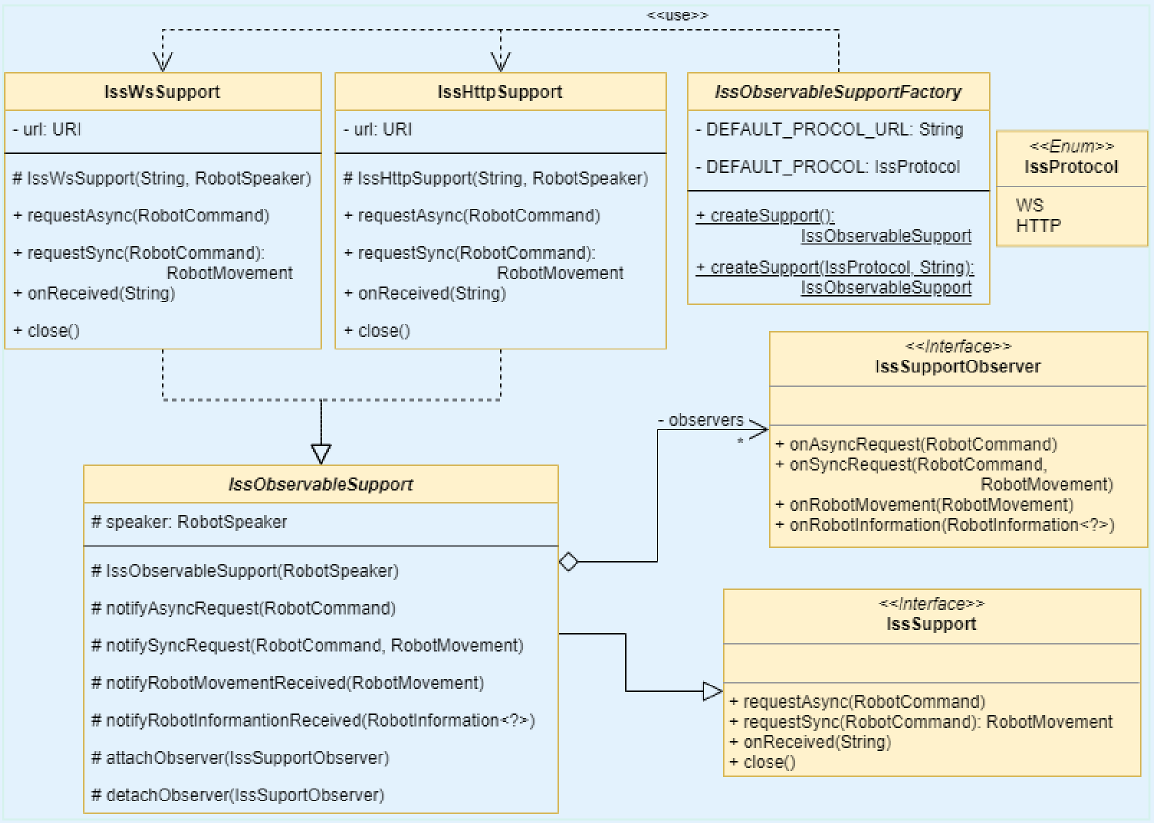
- **RobotCommand**: represents the cril command sendable via network;
- **RobotMovement**: represents the response received in cril language and then, in other words, the movement performed by the robot;
- **RobotInformation, SonardData, SensorData**: represent the data emitted by the the sonars and the sensors;
- **CommandFactory**: as specified in the ***Factory Pattern***, this static interface lets to easily create the objects of RobotCommand class;
- **RobotSpeaker**: interface that defines a component able to encode and parse the commands from/to the network (JSON) by/to the object worlds;
- **CrilRobotSpeaker**: implementation of RobotSpeaker for the entire application; notice that this class is a ***Singleton*** (this is the reason for the private constructor);
- **Move, MoveResult, Emitter**: utility enumeratives.

It's important to specify that **the objects of the class RobotCommand are only instantiable thanks to the Factory** meanwhile **only CrilRobotSpeaker can create the object of the classes RobotMovement and RobotInformation**.
In addition to this, notice that **RobotInformation is a parametric class**: the possible parametrizations are SonarData or SensorData and it's easily intuitive.
Finally, it's important to specify that the attributes **MOVES** and **MOVES_RESULTS** have relevant role in encoding and parsing commands/information and in a future development this two variable can be populated by some values contained in a configuration file for a most realiable way. For now, their values are the same:

```
private final static BiMap MOVES = HashBiMap.create(5);
private final static Map MOVE_RESULTS = new HashMap();
static {
        MOVES.put(Move.MOVE_FORWARD, "moveForward");
        MOVES.put(Move.MOVE_BACKWARD, "moveBackward");
        MOVES.put(Move.TURN_LEFT, "turnLeft");
        MOVES.put(Move.TURN_RIGHT, "turnRight");
        MOVES.put(Move.ALARM, "alarm");

        MOVE_RESULTS.put("true", MoveResult.SUCCEDED);
        MOVE_RESULTS.put("false", MoveResult.FAILED);
        MOVE_RESULTS.put("halted", MoveResult.HALTED);
        MOVE_RESULTS.put("notallowed", MoveResult.NOT_ALLOWED);
}
```

Support Level (package **it.unibo.resumableBoundaryWalker.supports**)

This package provides the necessary classes to realize communication between the application and the robot.



<<use>>
IssWsSupport
- url: URI
# IssWsSupport(String, RobotSpeaker)
+ requestAsync(RobotCommand)
+ requestSync(RobotCommand): RobotMovement
+ onReceived(String)
+ close()
IssHttpSupport
- url: URI
# IssHttpSupport(String, RobotSpeaker)
+ requestAsync(RobotCommand)
+ requestSync(RobotCommand): RobotMovement
+ onReceived(String)
+ close()
IssObservableSupportFactory
- DEFAULT_PROCOL_URL: String
- DEFAULT_PROCOL: IssProtocol
+ createSupport(): IssObservableSupport
+ createSupport(IssProtocol, String): IssObservableSupport
<<Enum>>
IssProtocol
WS
HTTP
<<Interface>>
IssSupportObserver
+ onAsyncRequest(RobotCommand)
+ onSyncRequest(RobotCommand, RobotMovement)
+ onRobotMovement(RobotMovement)
+ onRobotInformation(RobotInformation<?>)
- observers
*
IssObservableSupport
# speaker: RobotSpeaker
# IssObservableSupport(RobotSpeaker)
# notifyAsyncRequest(RobotCommand)
# notifySyncRequest(RobotCommand, RobotMovement)
# notifyRobotMovementReceived(RobotMovement)
# notifyRobotInformantionReceived(RobotInformation<?>)
# attachObserver(IssSupportObserver)
# detachObserver(IssSuportObserver)
<<Interface>>
IssSupport
+ requestAsync(RobotCommand)
+ requestSync(RobotCommand): RobotMovement
+ onReceived(String)
+ close()

- **IssSupport**: the interface that defines the behavior of the objects that carry out the communication with the robot;
- **IssObservableSupport**: an abstract class that realizes the ***Observer Pattern*** for the

IssSupport;

- **IssHttpSupport, IssWsSupport**: classes that extend IssObservableSupport and then respectively carry out the communication in Http and WebSocket protocols; these classes use the RobotSpeaker in order to encode and parse the messages;
- **IssObservableSupportFactory**: an abstract class that realizes the ***Factory Pattern*** for the supports;
- **IssSupportObserver**: an interface that defines the behavior of an Observer;
- **IssProtocol**: utity enumeratives for the factory.

It's very important to say that, also in this case, have been use two variable (**DEFAULT_PROTOCOL_URL** and **DEFAULT_PROTOCOL**) in the class IssObservableSupportFactory with any default values that can be populated thanks to a configuration file in a future development. Anyway, it's logical to say that the most suitable protocol is websocket than, for now, this is sufficient. We report the entire code of the factory:

```
public abstract class IssObservableSupportFactory {

        private static final IssProtocol DEFAULT_PROTOCOL =
IssProtocol.WS;
        private static final String DDEFAULT_PROTOCOL_URL =
"ws://localhost:8091";

    public static IssObservableSupport createSupport() throws
NoSuchIssProtocolException,
            DeploymentException, IOException, URISyntaxException {
        return createSupport(DEFAULT_PROTOCOL, DEFAULT_PROTOCOL_URL);
    }

    public static IssObservableSupport createSupport(IssProtocol protocol,
String url)
            throws NoSuchIssProtocolException, DeploymentException,
IOException, URISyntaxException {
        RobotSpeaker speaker = CrilRobotSpeaker.getSpeaker();
        switch (protocol) {
            case WS:
                return new IssWsSupport(url, speaker);

            case HTTP:
                return new IssHttpSupport(url, speaker);

            default:
                throw new NoSuchIssProtocolException(protocol + " not
available.");
        }
    }
}
```

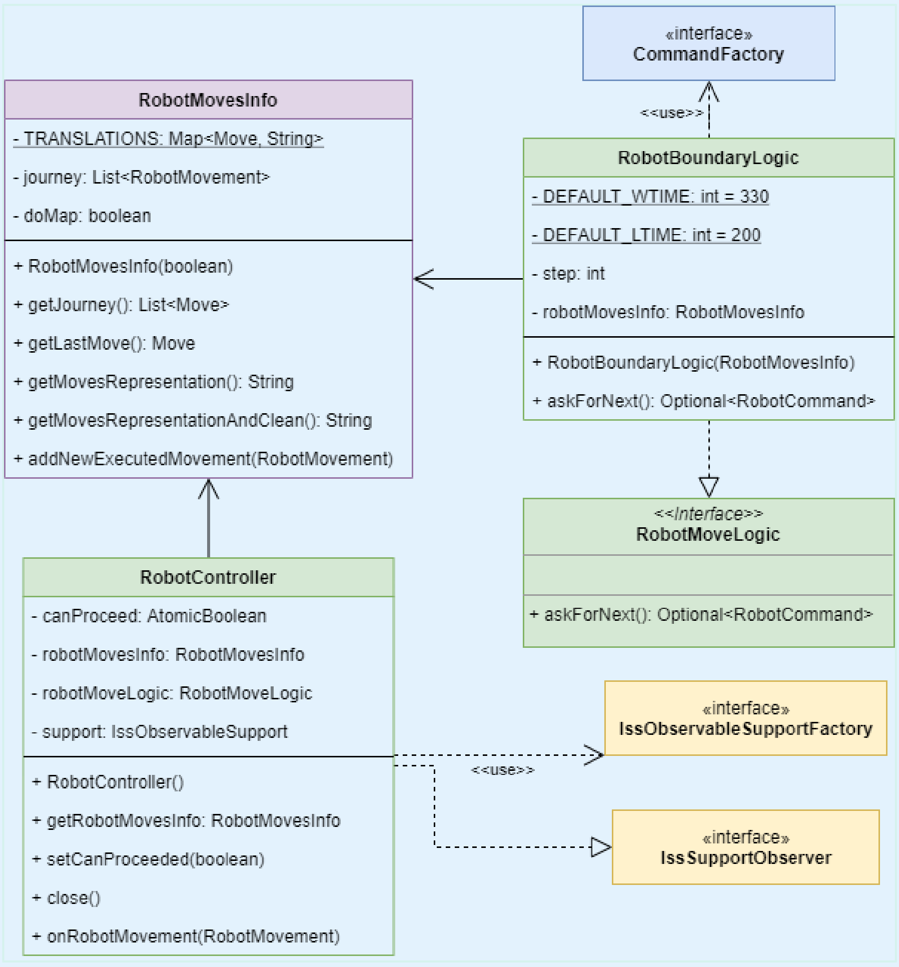
Even if it's not specified in the UML, in order to realize the "asynchronous" interaction with HTTP, **the class IssHttpSupport must implements the interface Runnable**, provided by the Java Standard Libraries. In addition to this, in order to manage the thread, it's needed to use a BlockingQueue and also an AtomicBoolean. For this reason we report the code of this class that use org.apache.HttpClient library:

```
public class IssHttpSupport extends IssObservableSupport implements Runnable{

    private static final int BUFFER_CAPACITY = 10;

    private CloseableHttpClient httpClient;
    private URI URL;

    private BlockingQueue<RobotCommand> commands;
    private AtomicBoolean service;

    protected IssHttpSupport(String URL, RobotSpeaker speaker) throws
URISyntaxException {
        super(speaker);
        this.httpClient = HttpClients.createDefault();
        this.URL = new URI(URL);

        this.commands = new ArrayBlockingQueue<>(BUFFER_CAPACITY);
        service = new AtomicBoolean(false);
        new Thread(this).start();
    }

    @Override
    public void requestAsync(RobotCommand command) throws  IOException{
        try {
            commands.put(command);
        } catch (InterruptedException e) {
            throw new IOException(e);
        }

        notifyAsyncRequest(command);
    }

    @Override
    public RobotMovement requestSync(RobotCommand command) throws
IOException {
        StringEntity entity     = new
StringEntity(speaker.encode(command));
        HttpUriRequest httppost = null;
        httppost = RequestBuilder.post()
                .setUri(URL)
                .setHeader("Content-Type", "application/json")
                .setHeader("Accept", "application/json")
                .setEntity(entity)
                .build();
        CloseableHttpResponse response = httpClient.execute(httppost);
        String jsonResponse = EntityUtils.toString( response.getEntity() );
        RobotMovement movement = speaker.parseMovement(jsonResponse);
        if(movement != null) {
            notifySyncRequest(command, movement);
        }

        return movement;
    }

    @Override
    public void onReceived(String message) {
        RobotMovement movement = speaker.parseMovement(message);
        if(movement != null)
            notifyRobotMovementReceived(movement);
    }

    @Override
    public void close() throws IOException {
        httpClient.close();

        service.set(false);
        try {
            commands.put(CommandFactory.createCommand(Move.MOVE_FORWARD,
-1));
        } catch (InterruptedException e) {
            throw new IOException(e);
        }
    }

    @Override
    public void run() {
        service.set(true);

        RobotCommand command = null;
        try {
            while(true) {
                command = commands.take();
                if(service.get() == false || command.getTime() <= 0) {
                    commands.clear();
                    return;
                }

                StringEntity entity     = new
StringEntity(speaker.encode(command));
                HttpUriRequest httppost = RequestBuilder.post()
                        .setUri(URL)
                        .setHeader("Content-Type", "application/json")
                        .setHeader("Accept", "application/json")
                        .setEntity(entity)
                        .build();
                CloseableHttpResponse response =
httpClient.execute(httppost);
                String jsonResponse = EntityUtils.toString(
response.getEntity() );

                onReceived(jsonResponse);

            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
```

About IssWsSupport, it's very easy to use the library *javax.websocket*.

Controller Level (package **it.unibo.resumableBoundaryWalker.controller**)

This package provides the controller of the entire application.


«interface»
CommandFactory
<<use>>
RobotMovesInfo
- TRANSLATIONS: Map<Move, String>
- journey: List<RobotMovement>
- doMap: boolean
+ RobotMovesInfo(boolean)
+ getJourney(): List<Move>
+ getLastMove(): Move
+ getMovesRepresentation(): String
+ getMovesRepresentationAndClean(): String
+ addNewExecutedMovement(RobotMovement)
RobotBoundaryLogic
- DEFAULT_WTIME: int = 330
- DEFAULT_LTIME: int = 200
- step: int
- robotMovesInfo: RobotMovesInfo
+ RobotBoundaryLogic(RobotMovesInfo)
+ askForNext(): Optional<RobotCommand>
<<Interface>>
RobotMoveLogic
+ askForNext(): Optional<RobotCommand>
RobotController
- canProceed: AtomicBoolean
- robotMovesInfo: RobotMovesInfo
- robotMoveLogic: RobotMoveLogic
- support: IssObservableSupport
+ RobotController()
+ getRobotMovesInfo: RobotMovesInfo
+ setCanProceeded(boolean)
+ close()
+ onRobotMovement(RobotMovement)
«interface»
IssObservableSupportFactory
<<use>>
«interface»
IssSupportObserver

- **RobotMovesInfo**: a class that represents the history of a journey and that memorizes the performed movements; this class was initially given by the consumer but has been adapted with the newer packages;
- **RobotMoveLogic**: the classes that implements this interface realize the bussiness logic for the sequence of the movements of the robot;
- **RobotBoundaryLogic**: the implementation of RobotMoveLogic for the application that makes the robot able to walk around the boundary of the room, as specified in

the requirements;

- **RobotController**: the controller of the application; we decide to create our controller class instead of use the given RobotInputController.

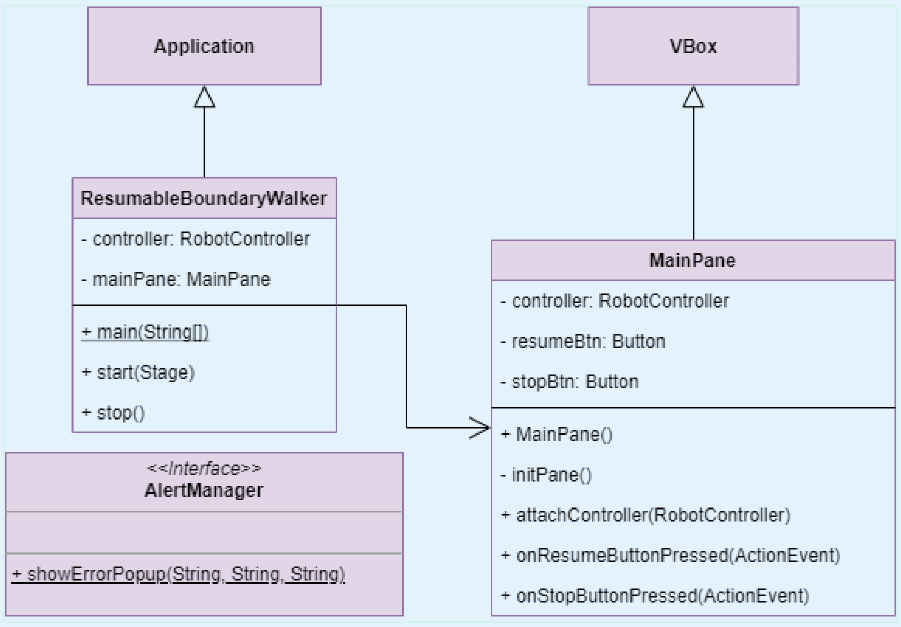
It's very important to say that **the controller is not a thread** thanks to the asynchronous interaction: the method *setCanProceeded(boolean canProceeded)* set the relative attribute of the class but also check if its value is changed. If **canProceeded** is passed from false to true, so the controller must use the support to send the next command but in the reverse case the controller must send the halt command. In addition to this, since the controller is an IssSupportObserver, after a movements, the same thread of the support immediately send in ansynchronous way the new command obtained from the RobotMoveLogic
Because of the importance of this class, we show his code:

```
public class RobotController implements IssSupportObserver {

    private RobotMovesInfo robotMovesInfo;
    private RobotMoveLogic robotMoveLogic;
    private IssObservableSupport support;

    private AtomicBoolean canProceed;

    public RobotController() throws NoSuchIssProtocolException,
DeploymentException,
            IOException, URISyntaxException {
        support = IssObservableSupportFactory.createSupport();
        canProceed = new AtomicBoolean(false);
        robotMovesInfo = new RobotMovesInfo(false);
        robotMoveLogic = new RobotBoundaryLogic(robotMovesInfo);

        support.attachObserver(this);
    }

    public void setCanProceed(boolean canProceed) throws IOException,
IllegalMovementException {
       boolean oldValue = this.canProceed.getAndSet(canProceed);

        if(canProceed == true && oldValue != canProceed) {
            Optional cmd = Optional.empty();
            while(!(cmd = robotMoveLogic.askForNext()).isPresent());

            support.requestAsync(cmd.get());
        }

        else if(canProceed == false && oldValue != canProceed) {
            support.requestAsync(CommandFactory.createCommand(Move.ALARM,
10));
        }
    }

    public void close() throws IOException {
        support.detachObserver(this);
        support.close();
    }

    @Override
    public void onAsyncRequest(RobotCommand command) {}

    @Override
    public void onSyncRequest(RobotCommand command, RobotMovement movement)
{}

   @Override
    public void onRobotMovement(RobotMovement movement) {
        robotMovesInfo.addNewExecutedMovement(movement);
        Optional&leqRobotCommand&geq cmd = Optional.empty();
        try {
             cmd = robotMoveLogic.askForNext();
        } catch (IllegalMovementException e) {
            e.printStackTrace();
        }

        if(canProceed.get()) {
            if(cmd.isPresent()) {
                try {
                    support.requestAsync(cmd.get());
                } catch (IOException e) {
                    e.printStackTrace();
                }
            } else {
                System.out.println("Boundary done. Jouney: " +
robotMovesInfo.getMovesRepresentationAndClean());
                canProceed.set(false);
            }
        }

    }

    @Override
    public void onRobotInformation(RobotInformation information) {}
}
```

We also specify that the attribute **TRANSLATIONS** of the class RobotMoves contains the string representation for all moves meanwhile the two variable **DEFAULT_WTIME** and **DEFAULT_LTIME** in RobotBoundaryLogic contain the default values for the time of the move forward and of turn left movements. In a future development their values could be taken from a configuration file in order to be most suitable.

Gui Level (package **it.unibo.resumableBoundaryWalker.gui** and class **it.unibo.resumableBoundaryWalker.ResumableBoundaryWalker**)

The package provides the required elements in order to build the GUI interface. We specify that we haven't used the given class **ConsoleGui** because it uses some deprecated methods. In addition to, instead AWT, we felt that ***JavaFX*** is most updated and suitable in order to realize the graphical interfaces. Then, the classes projected in the following UML uses the ***javafx library***.


Application
VBox
ResumableBoundaryWalker
- controller: RobotController
- mainPane: MainPane
+ main(String[])
+ start(Stage)
+ stop()
MainPane
- controller: RobotController
- resumeBtn: Button
- stopBtn: Button
+ MainPane()
- initPane()
+ attachController(RobotController)
+ onResumeButtonPressed(ActionEvent)
+ onStopButtonPressed(ActionEvent)
<<Interface>>
AlertManager
+ showErrorPopup(String, String, String)

- **MainPane**: the class that realizes the main pane of the graphical interface; it contains the two buttons requested by the requirements; as specified in JavaFX documentation, the two methods **onResumeButtonPressed** and **onStopButtonPressed** are the ***Action Listeners*** of the two button that invoke the relative method of the controller;
- **AlertManager**: a simple static class that make showing popup more easier;
- **ResumableBoudaryWalker**: the main class of the application that paints the graphical interface also initializing all components (see ***javafx.application.Application***);

Testing

In order to test the requirement of the boundary and of the production of the final string, we have produced a JUNIT test that uses the maded classes. To perform the test, it's needed to access to the variables **canProceeded** and **robotMovesInfo** of the RobotController, so we have added three new methods to this class: **getIsProceeding(): boolean**, **getIssSupport(): IssObservableSupport** and **getLatJouney(): String**.

```
class IssSupportObserverForTest implements IssSupportObserver {

    private static Map TRANSLATIONS = new HashMap<>();
    static {
        TRANSLATIONS.put(Move.MOVE_FORWARD, "w");
        TRANSLATIONS.put(Move.MOVE_BACKWARD, "b");
        TRANSLATIONS.put(Move.TURN_LEFT, "l");
        TRANSLATIONS.put(Move.TURN_RIGHT, "r");
    }

    private String moves;

    public IssSupportObserverForTest() {
        this.moves = "";
    }

    public String getMoves() {
        return this.moves;
    }

    @Override
    public void onAsyncRequest(RobotCommand command) {}

    @Override
    public void onSyncRequest(RobotCommand command, RobotMovement movement) {}

    @Override
    public void onRobotMovement(RobotMovement movement) {
        moves += TRANSLATIONS.get(movement.getMove());
    }

    @Override
    public void onRobotInformation(RobotInformation information) {}
}

public class AppTest {

    private RobotController controller;
    private IssSupportObserverForTest observer;

    @Before
    public void init() throws Exception {
        controller = new RobotController();
        observer = new IssSupportObserverForTest();

        controller.getSupport().attachObserver(observer);
        controller.setCanProceed(true);
    }

    @Test
    public void testDoBoundary() throws InterruptedException {
        while(controller.isProceeding() == true)
            Thread.sleep(1000);

        String moves = controller.getLastJouney();
        String obtainedFromTest = observer.getMoves();

        assertEquals(moves, obtainedFromTest);
        assertTrue(moves.matches("w+lw+lw+lw+l"));
    }
}
```

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 student
Name: Luca Marchegiani
Email: luca.marchegiani3@studio.unibo.it
Git Repo: https://github.com/LM-96/MarchegianiLuca