Project activity of systems engineering software
Project "Line follower"

Alessio Tonioni

Alma Mater Studiorum – University of Bologna
viale Risorgimento 2, 40136 Bologna, Italy
alessio.tonioni@studio.unibo.it

1 Introduction

The aim of this project it's to realize an independent controller that controls a robot and makes it capable of following a line on the ground. Furthermore the focus of the work is on the property of the software developed and how to organize the work with the purpose of obtaining a piece of reliable and reusable code.

2 Vision

The development of the controller should be fully automatable given the fundamental parameters of the robot, so the controller should be as general purpose as possible and should not be bonded to a specific type of robot or to a sensors configuration. After the initial development new controllers should not be implemented by code, but automatically with the help of simplified language capable of describe exhaustively the robot.

3 Goals

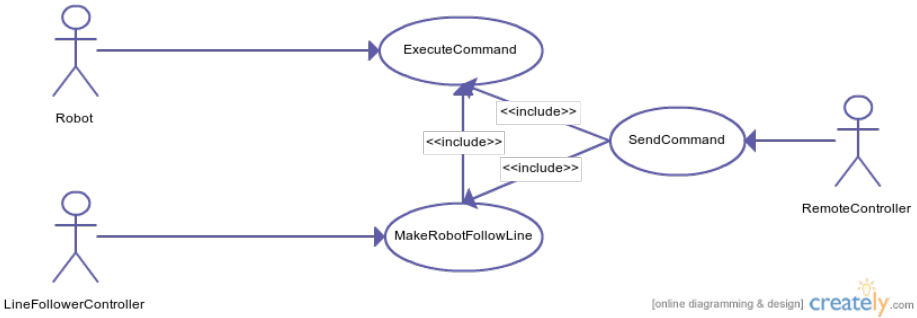
Realize a piece of software capable of making a robot able to follow a line on the ground. Furthermore this program must be reusable and extensible in future development.

4 Requirements

Realize an independent controller to make a robot able to follow a line on the ground perceptible with on-board sensors.
Normally the robot is remote-controlled from a remote device, but when requested the line's follower takes the control of the robot and drives it till new commands are given.
At any given moment during the execution of the controller new commands could be sent from the remote device, when this happen the controller stop working and the robot must returns to remote-controlled only.

5 Requirement analysis

5.1 Use cases



5.2 Scenarios

ExecuteCommand:
Actor: robot
Post-condition: the robot is executing a command.

Main scenario:

- The robot is asked to execute a simple command
- The robot starts to execute the command and continues till a new one is executed

MakeRobotFollowLine:
Actor: Line follower controller
Post-condition: the robot is moving autonomously and it's following a line on the floor
Main scenario:

- The robot is asked to start following a line
- The robot starts to follow the line and continues till a new command are requested

ReceiveCommand:
Actor: remote controller
Pre-condition: the remote controller knows how to communicate with the robot
Post-condition: the robot executes the command sent.
Main scenario:

- A command or the request to start a line follower controller is sent from the remote controller.
- The robot starts to execute what is requested to do.

5.3 Glossary

Term	Meaning
Command	Basic action that the robot it's capable of interpret and execute.
Independent Controller	Autonomous entity capable of control a robot giving it command.
Line	Mark perceptible with on-board sensors.
Remote device	Entity independent from the robot able to communicate with it. It can ask to a certain robot the execution of a command or the start of a line follower controller.
Robot	Entity capable of interpret command.

5.4 (Domain)model

The domain it's made of at least two physically separated macro entity:

Remote-device:
Entity capable of connect in some way to the robot and send it command, it could be formalized with this interface.

```
1 package interfaces;
2
3 public interface RemoteDevice {
4   void sendCmd(String msg);
5   void setsCommandReceiver(ICommandReceiver receiver);
6   ICommandReceiver getCommandReceiver();
7 }
```

/home/alessio/git/AttivitaProgettuale/ModelInterfaces/src/interfaces/RemoteDevice.java

Robot:
The physical entity robot could be decomposed in three sub-system:

- **Executor:** interpret of commands already realized, it could be formalized with the given interface:

```
1 package it.unibo.iot.robot;
2
3 import it.unibo.iot.models.robotCommands.IRobotCommand;
4
5 public interface IRobot {
6
7   public void execute(IRobotCommand command);
8
9 }
```

/home/alessio/git/AttivitaProgettuale/it.unibo.iot.robot/src/it/unibo/iot/robot/IRobot.java

– **Command Receiver:** entity with the task of receiving messages from the remote devices, interpret it and translate them in command for the robot or the line follower. It could be formalized with this interface.

```
package interfaces;

import it.unibo.iot.robot.IRobot;

public interface ICommandReceiver {
  String receiveMessage();
  void executeRequest(String message);
  void setRobot(IRobot robot);
  IRobot getRobot();
  void setLineFollower(ILineFollower follow);
  ILineFollower getLineFollower();
}
```

/home/alessio/git/AttivitaProgettuale/ModelInterfaces/src/interfaces/ICommandReceiver.java

– **Line follower:** entity that when started takes control of a robot with the purpose of making it follow a line on the ground, it must be stopped when requested from the outside and should be able to communicate with the sensors on the robot. If possible this component should be reusable across different robots or sensors configuration. It could be formalized with this interface:

```
package interfaces;

import it.unibo.iot.robot.IRobot;

public interface ILineFollower {
  void setRobot(IRobot robot);
  IRobot getRobot();
  void start();
  void terminate();
}
```

/home/alessio/git/AttivitaProgettuale/ModelInterfaces/src/interfaces/ILineFollower.java

This three entities could be active at the same time: the "Command Receiver", for example, is always listening for incoming messages meanwhile the robot is executing a command or the controller is following a line. The three sub-systems are all inside the robot system because "executor" and "line follower" must communicate with external device on board on the robot: engine and sensors respectively; Command receiver instead must communicate with executor.

5.5 Test plan

In order to introduce automatic test cases a modified robot must be modelled:

```java
package test;

import it.unibo.iot.models.robotCommands.IRobotCommand;
import it.unibo.iot.robot.IRobot;

public interface MyIRobot extends IRobot{
  boolean isOnLine();
  IRobotCommand getCurrentCmd();
}
```

/home/alessio/git/AttivitaProgettuale/ModelInterfaces/src/test/MyIRobot.java

Now the interface of the model can be tested:

```java
package test;

import static org.junit.Assert.*;
import interfaces.ILineFollower;

import org.junit.Before;
import org.junit.Test;

public class LineFollowerTest {
  private MyIRobot testRobot;
  private ILineFollower follower;

  @Before
  public void setUp() throws Exception {

  }

  @Test
  public void testRobotSet(){
    follower.setRobot(testRobot);
    assertEquals(follower.getRobot(),testRobot);
  }

  @Test
  public void testLineFollowing() throws
    InterruptedException {
    follower.setRobot(testRobot);
    follower.start();
    Thread.sleep(10000);
    assertTrue(testRobot.isOnLine());
    follower.terminate();
  }

}
```

/home/alessio/git/AttivitaProgettuale/ModelInterfaces/src/test/LineFollowerTest.java

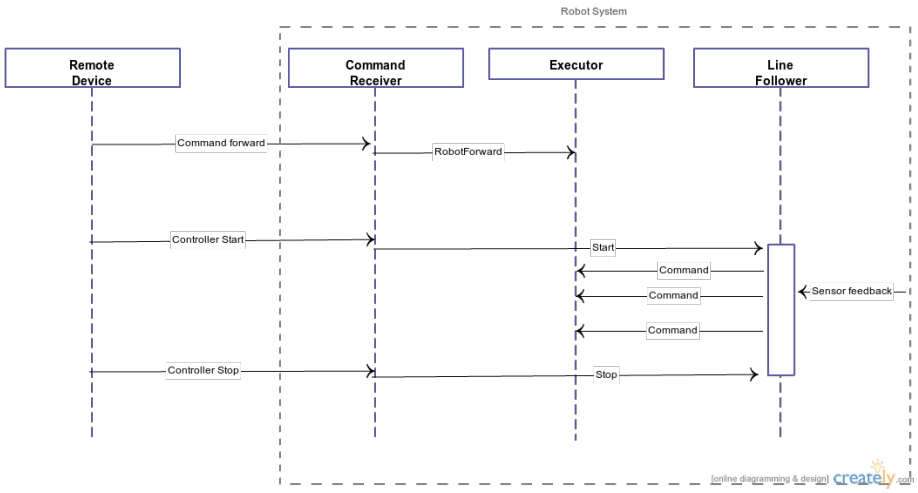
```java
package test;

import static org.junit.Assert.*;
import interfaces.ICommandReceiver;
import interfaces.ILineFollower;
import interfaces.RemoteDevice;
import it.unibo.iot.models.robotCommands.RobotForward;

import org.junit.Before;
import org.junit.Test;

public class RemoteDeviceTest {
  private ICommandReceiver receiver;
  private MyIRobot testRobot;
  private RemoteDevice remote;
  private ILineFollower follow;
  @Before
  public void setUp() throws Exception {
    receiver.setLineFollower(follow);
    receiver.setRobot(testRobot);
    remote.setsCommandReceiver(receiver);
  }

  @Test
  public void testGetterSetter() {
    assertEquals(remote.getCommandReceiver(),receiver);
    assertEquals(receiver.getLineFollower(),follow);
    assertEquals(receiver.getRobot(),testRobot);
  }

  @Test
  public void testsimpleCommand(){
    remote.sendCmd("cmd forward 50");
    receiver.executeRequest(receiver.receiveMessage());
    assert(testRobot.getCurrentCmd() instanceof
    RobotForward);
  }

  @Test
  public void testLineFollowing() throws
    InterruptedException{
    remote.sendCmd("cnt start");
    receiver.executeRequest(receiver.receiveMessage());
    Thread.sleep(1000);
    assert(testRobot.isOnLine());
  }

}
```

/home/alessio/git/AttivitaProgettuale/ModelInterfaces/src/test/RemoteDeviceTest.java

6 Problem analysis

6.1 Logic architecture

The software system is composed of at least two physically separated entities that must communicate, the semantic of the communication its asynchronous and mono directional. A typical message exchange of the whole system could be pictured with this schema:



All the messages are asynchronous, the content of the message between "Remote device" and "Command receiver" should conform to a certain grammar in order to make the communication possible.
The messages directed to the "executor" should conform to the given IRobot interface, for example each message could correspond to a proper call of the "execute()" method of the IRobot.
The messages of start and stop from the "command receiver" to the "line follower" could be seen as creation and destruction of the "line follower", furthermore this follower must be able to receive incoming messages from the sensors so it must communicate with them.
As seen all the interaction between the remote device and the robot system are transmitted through the "command receiver" that it's the only point of communication between the robot system and the outside world, the communication is mono directional, so there are no feedbacks from the robot on the correct execution of the requested action.
The sensors' feedbacks are produced independently from the rest of the system, this type of messages are asynchronous and are already implemented using an

observer pattern to be notified when new measurement are available.

From this representation we can see that the "executor" is a completely passive entity capable only of execute commands when requested, instead the "line follower" is an active entity with a limited life span that could be created and destroyed at any given time, at this level of abstraction we are not interested on how the controller does it's job, the only thing it must guarantee is that when activated the robot start to follow the line. The "Command receiver" is an interpret that translate the request of the remote device in command for the "executor" and the "line follower", it must be created at the startup of the system and must remain active till the system is turned off.
Every "command receiver" must have an unique identifier so that a remote device can only send message to a specific one. In this architecture is possible that more than one remote devices send messages to the same command receiver, in this case there's no priority and the execution of the messages is FIFO.

6.2 Abstraction gap

The infrastructure to control the robot engines and sensors is already developed, so there should be no problem for the interaction with physical devices. The communication between the entities could be implemented using standard tcp/ip protocol in order to be cross platform and scalable. The main abstractions gap are the inability to control directly the engine of the robot in order to achieve a higher precision when following a line and the absence of a way to tune the line follower controller for a certain robot in a fully automated way. Some experimental approach using genetic computation will be presented later, but the result are still not perfect.

7 Project

7.1 Structure

The overall structure of the system conform to the one presented in the logic architecture part, however there's some little clarification on how the single entities are implemented.

Command Receiver −> listenerController: project it.unibo.RobotServer
To achieve the maximum independence from the technology used in different systems, the command receiver is realized as a simple tcp server, and the messages are so composed:

- 1 byte for the size of the message
- A string encoded using UTF-8 standard

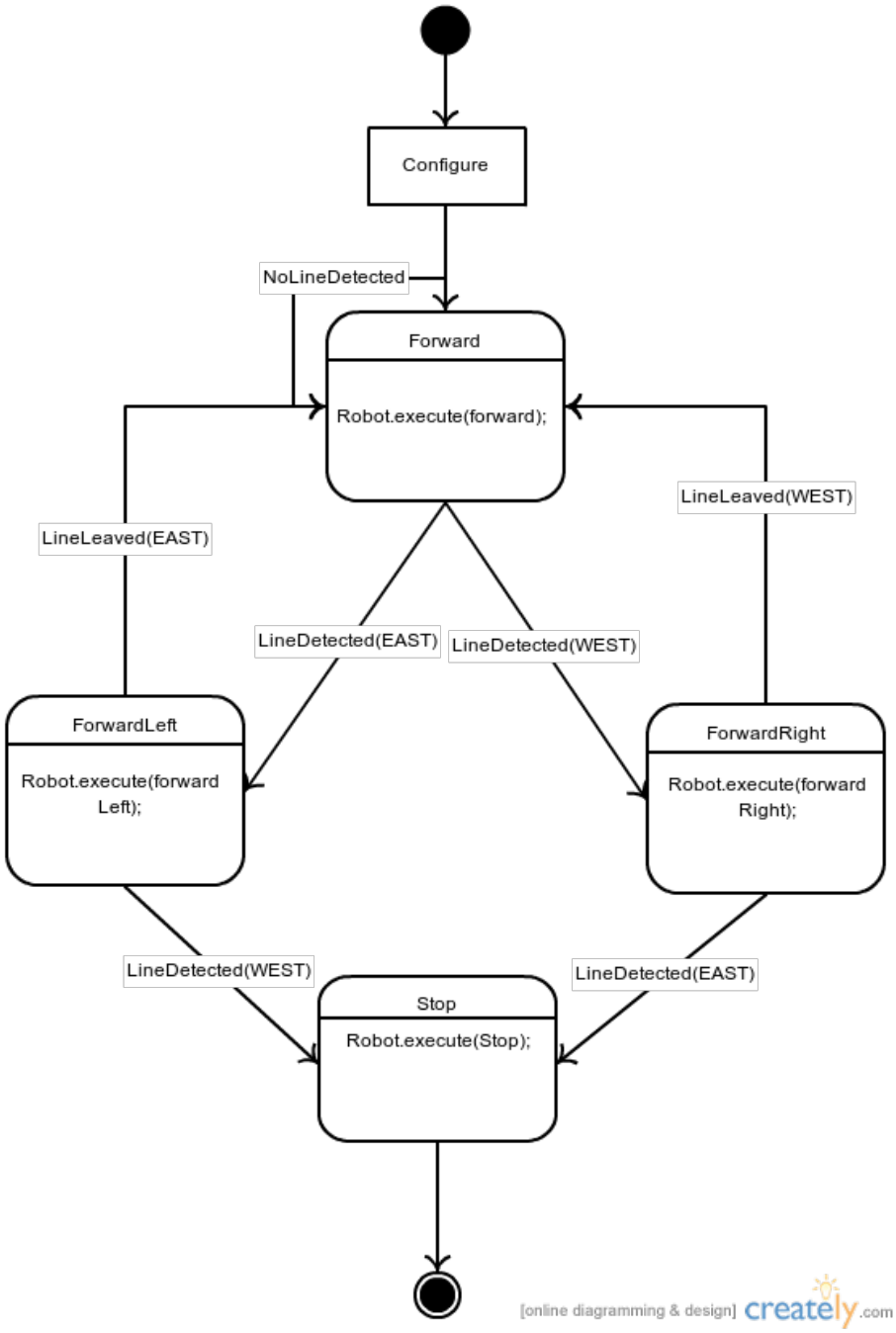
In order to be correctly interpreted the string must follow this grammar:

```
SCOPE: MSGTYPE
MSGTYPE: COMMANDMSG | CONTROLLERMSG
COMMANDMSG: "cmd " COMMANDTYPE SPEED
CONTROLLERMSG: "cnt " CNTTYPE SPEED FORWARD
COMMANDTYPE: "forward" | "backward" | "left" | "right" | "
    fordwardRight" | "forwardLeft" | "backwardRight" | "
    backwardLeft" | "stop"
CNTTYPE: "PID" | "StateBiLine" | "StateMonoLine" | "PIDFinale"
SPEED: integer between [0,100]
FORWARD: "true"|"false"
```

The "command receiver" uses an external component, implementation of **ICommandExecutor**, to make the physical robot execute commands or to start the "line follower" controller, in this way some parts of the server are reusable for different physical robot.

Remote Device: project RemoteControlRobot
For testing purpose the remote device is implemented as a simple Windows phone app, the only thing it must do is to connect to the tcp server on the robot and send to it message composed as described above.

LineFollower –> ILineFollower project it.unibo.lineFollower
This component must be able to control autonomously the robot but at the same time must be stoppable, for this reason the ILineFollower interface extends Runnable so every controller can be instantiated as a thread that iteratively calculates the corrective commands to make the robot follow a line till someone kills it using the method terminate().
The controller works above the Robot so the rebot isn't aware of following a line.
The core of the project is the implementation of the ILineFollower interface for this reason a lot of different possibilities were considered during the development. The aim of this component is to make the robot able to travel inside two lines that act like barrier to delimit the correct track and to make the motion of the robot as smooth as possible: if it's possible the robot must not stop but always continue along its path applying the littlest possible correction.
The robot at the user level of abstraction was capable only of execute an "IRobotCommand" so only a limited set of movements were possible, for this reason the first instance of controller was a simple state machine that applies a corrective command according to the data from the sensors. The behavior of the component could be pictured with this schema.



This component is fully reactive and change its state only when new data are sent from the sensors, so it is not aware of the time and from how long the robot

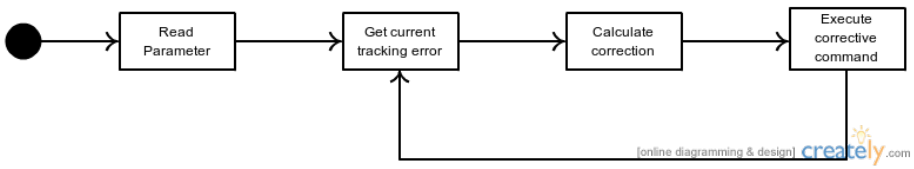
is traveling in the wrong direction. However the results were not really good: the robot tends to react slowly and without sufficient effort so it easily losts the right path.

From this first experiment it appears necessary to modify the interface "IRobot" to have a direct control of the robot's motors in order to perform all the possible corrective command. A new interface is defined: "IDifferentialDriveRobot" that extends "IRobot" and adds a method to control the robot via "IWheelCommand" so giving the speed of each of the robot's wheel.

```
1 package it.unibo.iot.robot;
2
3 import it.unibo.iot.models.wheelCommands.IWheelCommand;
4
5 public interface IDifferentialDriveRobot extends IRobot {
6
7   public void execute(IWheelCommand wheelCommand);
8 }
```

/home/alessio/git/AttivitaProgettuale/it.unibo.iot.robot/src/it/unibo/iot/robot/IDifferentialDriveRobot.java

With this new interface we have full control on the robot, so different degrees of correction can be applied in different situation. Now we need a component capable of determine the effort of the corrective action from the sensors data and the time, a simple state machine is not enough, something more expressive and modular is needed.
Based on previous knowledge arising from the course of automatic controls an experimental solution was developed using a standard PID controller.
A PID is a standard type of closed chain controller that is used to control autonomous system, it calculates an error value as the difference between a measured process variable and a desired setpoint. The controller attempts to minimize the error by adjusting the process through use of a set of manipulated variable. For more information see Bibliography.
In the implementation for the line following the setpoint is 0 and the error ("tracking error") indicates whether the robot is careening to the right or left from the ideal track. The controller is constantly looping so has knowledge of the time in the form of number of consecutive iterations.
The behavior of the controller can be pictured as follow:



The tracking error as said above is obtained from the sensor's data and the corrective command is a certain "IWheelCommand", the core of the controller is the "calculate correction" that calculate the right effort of the corrective command based on the current error and the past history. More specifically the effort is

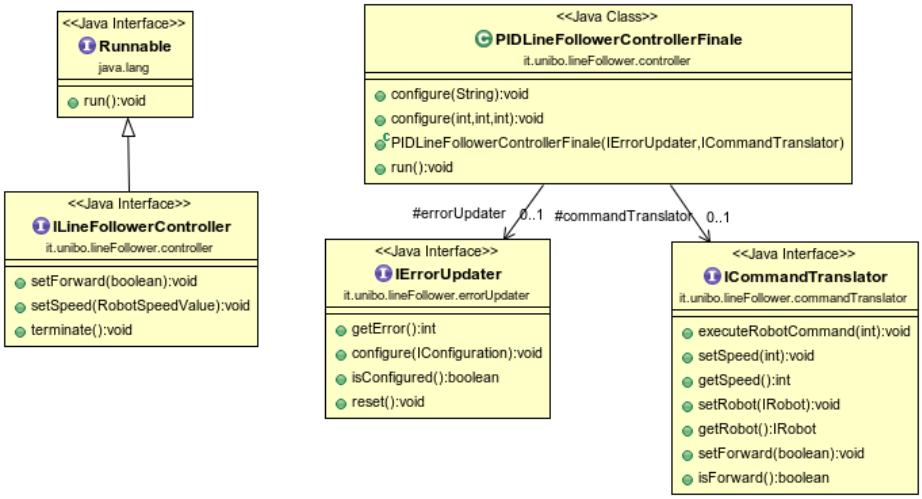
obtained from the current error (CE), the derivative of the error (DE) and the integral of the error (IE) multiplied by the respective constants (named: P,I,D). The correction is obtained as follow:

$$Correction = P * CE + I * IE + D * DE$$

and is then translated in a variation on the default speed for the wheel of the robot, so in a new "IWheelCommand", that remain in execution till the next loop of the controller is completed.
A PID controller can be seen as the union of three different controllers: Proportional, Integral and Derivative that corresponds to the three constants said above. Every controller has a specific role:

- **Proportional:** higher values of the constant make the controller react more strongly to errors. It's the basic controller, when there's a error try to fix it with a proportional effort.
- **Integral:** higher values of the constant make the controller react more strongly when errors are presents from long time. In combination with the other is used to obtain a smoother drive for the robot.
- **Derivative:** higher values of the constant make the controller react faster when the errors happen. Is useful when we want a controller that react quickly when an error appear.

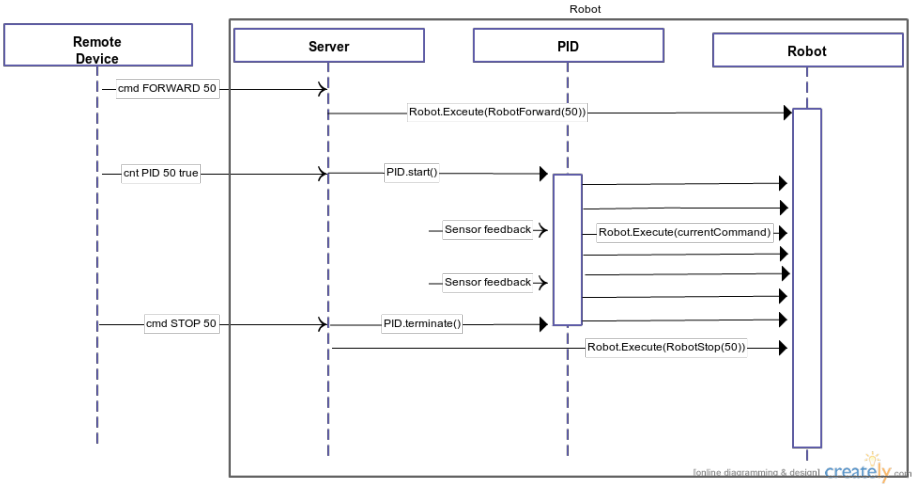
The controller PID is therefore very modular because it is possible to use only some of the three components by setting the related constants to 0, for example we can obtain a fully proportional controller setting I and D to 0.
The aim of this type of controller is to complete a loop as fast as possible in order to calculate always the best turn based on fresh data, but this can be a problem if the controller works faster than the actuators, so a modular delay can be introduced between each iteration in order to manipulate how the controller behave.
This behavior can be implemented as a simple "main" program and works really well as long as the controller is properly tuned with the three constants, but is not extensible or reusable: if the sensor configuration or the robot change, everything must be rewritten.
The first thing to do was to remove the three constants and the delay time from the code and put them in an external file, so they can be easily changed without the need to stop the robot and deploy a new jar. Then a new architecture for the controller was developed based on the principle of the separation of the responsibility: the communication with the sensors and the command execution were decoupled from the controller loop and assigned to two independent pieces of software described by the **IErrorUpdater** and **ICommandExecutor** interfaces respectively.



The IErrorUpdater calculate the tracking errors when the sensor's give new feedback, it can be queried using the getError() method in order to obtain the current error. Before start using the configure method must be called. The error calculated from an implementation of IErrorUpdater must be an integer between [-100,100] where 0 means no error, -100 totally wrong on the left and +100 totally wrong on the right. Through this measure of the error the **PID** controller generates another integer which represents the correction that must be applied to the robot to make it return on the correct track, is now the turn of the ICommandExecutor to take this integer and translate it in command that the robot is capable of execute.
The behavior of the PID controller can now be modified with three degrees of freedom: the constants determine how fast and how strongly the controller react to errors, the implementation of IErrorUpdater determines how to calculate this error and so what is considered an error and finally the implementations of ICommandExecutor determines how the correction is physically executed.
Using this structure the controller can be reused with any sets of sensors or robot and to perform several different tasks, for example it could be used to run a robot inside an aisle just creating different implementation of the IErrorUpdater.

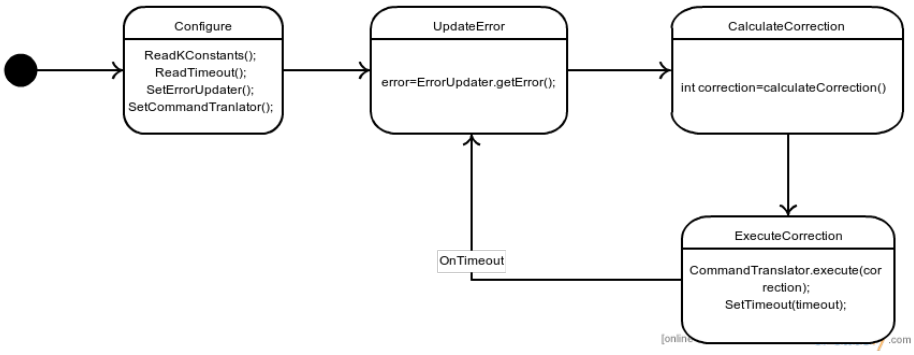
7.2 Interaction

The communication between the remote device and the command executor is realized using standard tcp socket. Tcp is preferred over udp because ensure the correct order of arrival of the commands.
All the interaction inside the robot entities instead are implemented as simple procedure call.
A typical interaction of the system can be pictured like this:



7.3 Behavior

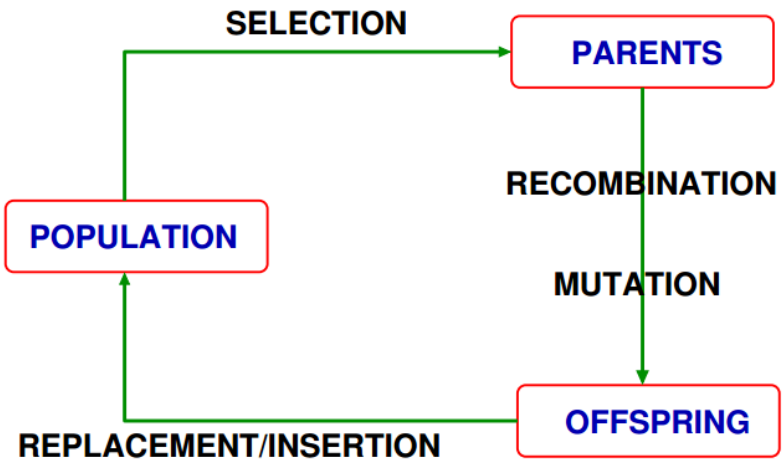
The overall behavior of the final implementation of pid controller follow this state machine



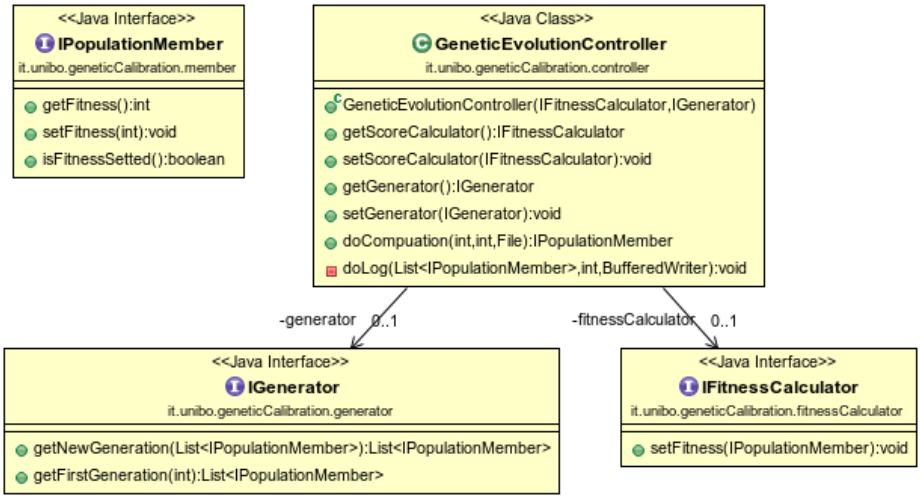
8 Implementation

Given the overall structure the implementation is quite simple and does not presents difficult point. As said above the structure is very reusable because of the modularity and adaptability, but it must be properly **tuned** before starting. Tuning corresponds to properly sets the 3 constants of the PID controller in order to make it able to control a specific robot.

Before using a controller there must always be a phase of long testing to find the proper constant for the robot and the sensors' configuration. In order to try to automatize this task is been tested a method based on Evolutionary computation: using a simulator and a lot of iterations we try to find the best possible constants to control that particular simulated robot. The more the simulated robot behavior is close to the real one the more this technique can be useful.



The genetic computation program starts from a population of individuals, in our case different combination of PID constants, and at each iteration gives at all of them a score, called fitness, based on how well they behave. From this population a new one is generated recombining the best current solutions and adding some random mutation, the new generation created should perform better than the one before and the iteration could restarts. After a certain number of iteration we should obtain a good individual among the population, so a good set of constants.
In order to implement this algorithm in the most general way possible the logic of population creation and score calculation are relegated to two external components: **IGenerator** and **IFitnessCalculator**.



The **IGenerator** gives the initial population to the algorithm and at each genetic loop is responsible for the phase of selection recombination and mutation that generates the new generation. The behave of the algorithm and the speed with which converges towards an optimal solution depends strictly from this component.
The **IFitnessCalculator** has the task of associating to each individual a score counter based on how well they perform, so its responsible for the correctness of the solution.
For the PID calibration the generator implementation create the first generation totally with random values, then at each iterations takes the parents for the recombination among the fittest individuals of the previous generation, in the new generation is also maintained the fittest individual of the previous one in order to speed up the algorithm. The fitness calculator is instead based on a simple simulator of the robot: the behave of the individual, so of the PID controller configured with its set of constants, is tested in a simulated environment and a score is given to its performance. The simulator uses the implementation of PID controller presented above but instead of controlling a real robot, it controls a simulated one based on the simple differential drive law of physics.

9 Testing

All the system was tested using a differential drive robot with two line sensors and a windows phone for the remote device. The results are quite good, but the constants calculated using the genetic computation needs a refinement by hand probably because of the too simplistic simulation.

10 Deployment

The system is deployed as a single .jar for the robot and an application for the remote device. In order to work properly the jar needs also a series of configuration file both for how to interact with sensors and motors and for how to configure the controller.

11 Conclusion

12 Information about the author

Alessio Tonioni
alessio.tonioni@studio.unibo.it

Photo of the author

13 Bibliography:

– **PID controller** http://en.wikipedia.org/wiki/PID_controller