Crossroads
The ACM Student Magazine

# ART - the Abstract Robot Toolkit

by ***Stephan Jätzold***

## Introduction

Lego Mindstorm systems have captured the imagination of children, hobbyists, and even serious programmers all around the Internet. This article is about a new object-oriented programming system developed in Java and designed to write programs for controlling robots *independently* of the actual hardware: the Abstract Robot Toolkit (ART).

### Outline

This tutorial presents practical examples and is not meant to be a thorough overview of the API. The article demonstrates ART's basic functionality in three steps:

1. "Hello, Robot!" -- A simple example showing how to turn a motor on.
2. `DriveTrain` -- A reusable chassis for mobile robots.
3. `Trusty` -- An obstacle-avoiding mobile robot using a `DriveTrain`.

### Overview

The concept of combining a construction set with a computer was actualized by FischerTechnik in 1984 with the "Universal Interface" and again in 1997 with the "Intelligent Interface." The Intelligent Interface already had a microprocessor enabling it to run independently of a PC. However, the FIscherTechnik systems were overshadowed by the Lego Mindstorm systems, which were introduced in 1998. Possibly the biggest fan site for Lego is the Lego Users Group Network (**www.lugnet.com**). There, especially in **lugnet.robotics** and its subgroups, a wealth of information about Lego Mindstorms can be found.

The RCX is the programmable brick from Lego Mindstorms. It is microprocessor-based and is capable of controlling motors and sensors. Unlike the FischerTechnik Intelligent Interface, the RCX's operating system (its firmware) is documented and can be replaced. Consequently, the RCX is an open system. Several programming environments are available for it over the internet.

Several programming systems exist for interfacing with robots from Lego and FischerTechnik. Some are graphical systems running on a PC (**RCX-Code**, **RoboLab**, **LLwin**), some are drivers for conventional programming languages but still need a PC (**Spirit.ocx**, **Fishface**), and for the RCX brick from Lego there are also some programming languages available which are executed by the brick itself -- either with the standard firmware from Lego (**NQC**, **Mindscript**) or with their own replacement firmware (**legOS**, **pbForth**, **leJOS**).

### Motivation

ART was developed to provide a universal JAVA API for programming robots. Several JAVA robot languages had already been developed, including Axel Schreiner's Java-driver for FischerTechnik [**9**] and Dario Laverde's driver for the RCX [**8**]. The RCX JAVA API is mostly a wrapper for the communications protocol of the RCX; you still had to know (and use!) the raw byte-codes. The ft-package by Axel Schreiner already had objects for representing motors and sensors but was still focused on the Intelligent Interface. The two systems are radically different and provide access to only the basic capabilities of their respective hardware interface.

The lack of a single, easy to use, uniform driver for two kinds of robot hardware motivated the creation of ART.

**Figure 1:** Two robots built with different hardware but both able to move around and avoid obstacles in the same way.

At first blush, the Abstract Robot Toolkit is just another driver that uses the JAVA language and runs on a stationary PC. However, its object-oriented design and ease of use and re-use set it apart from others. In addition, ART abstracts away the actual robot hardware so the same code controls multiple robot hardware types.

### Limitations

Programming a simple robot shows how to use the basic classes of the Abstract Robot Toolkit (ART). The current implementation has drivers for Lego and FischerTechnik. The examples can also be used with a "virtual" driver requiring no robot hardware. To some degree the system permits plugging software components used to control a robot just like the lego-bricks itself.

A program using ART runs on a stationary PC because it needs the Java Runtime Environment and the Java Communications API. This has the drawback of a slow infrared connection to Lego. Also, the abstraction layer of ART has not been tuned for speed and surely offers no realtime capabilities. But, it is not difficult to use, and with the Intelligent Interface from FischerTechnik it is fast enough for many applications. The Lego brick can be used if speed does not matter that much.

By using a stationary PC to control a robot, one has access to greater processing power. A wide range of already existing code can be used and user interaction or integration in existing applications is easily accomplished. However, a communications channel between the PC and the robot hardware has to be employed which can sometimes be slow, as in the case of the IR-Link of the RCX. Also, the mobility of the robot can be reduced because of a connecting wire like the serial cable of FischerTechnik's Intelligent Interface, or because of the range of a wireless link. On the other hand, if the Robot is controlled locally, by running programs on its own computer, processing power is limited if size and power-consumption matter.

When using the original Lego firmware, one communications cycle is not only slow (about 100 ms), but in each cycle only one sensor value can be retrieved, or one of three parameters of a motor (on/off/float, power, direction) can be manipulated. With FischerTechnik, all information about the inputs and outputs can be exchanged in one cycle. There are ways to overcome this problem with Lego. For example, one can do this by implementing a seperate communications protocol on top of the set/getVariable mechanism (see [**5**]), or by using a different firmware. LegOS might be a good candidate because it already has a built-in communications stack for the IR-Link called LNP [**1**], but this has not been explored for ART thus far.

A tutorial explaining how to implement a new ART driver for robot hardware is available in German [**4**], see **www.jaetzold.de/art/art.pdf**.

### "Hello, Robot!"

A very common example when introducing a new programming language is "Hello, World!" [**6**]. For robots, this could be likened to turning a motor on. Programmed with ART, this looks as follows:

```
package de.jaetzold.art.examples;

import de.jaetzold.art.RobotInterfaceFactory;
import de.jaetzold.art.RobotInterface;
import de.jaetzold.art.ActuatorPort;
import de.jaetzold.art.Motor;

public class HelloRobot {
    public static void main(String[] argv) throws InterruptedException {
        Motor motor = new Motor();

        // get an interface to the actual hardware
        RobotInterfaceFactory factory = new RobotInterfaceFactory();
        RobotInterface hardware = factory.getInterface();

        // get a port and connect the motor to it (match hardware!)
        ActuatorPort port = hardware.getActuatorPorts(0);
        motor.connectWith(port);

        // turn it on
        motor.on();

        // delay and quit
        Thread.sleep(1000);
    }
}
```

### How to execute `HelloRobot`

Compilation requires the Java Development Kit version 1.2 or later; execution requires the Java Runtime Environment with the Java Communications API (JavaComm) installed. The class files making up ART must be available. For Java and the Java Communications API see **http://java.sun.com**. or take the JAVA implementation of your choice. Concerning the speed of the JAVAComm-implementation I have had the best experiences using the current implementation from IBM running on Linux or Windows. The newest version of ART can be downloaded from **http://www.jaetzold.de/art/**. The source is compiled by a call to javac, for example:

```
javac -classpath <some-path>:art.jar HelloRobot.java
```

which creates a file called "HelloRobot.class" in the same directory as the .java file. For this example, it has been assumed that the file "art.jar" is in the current directory. If it is not, its path must be specified, too. To execute this class, the fully qualified class name is passed as an argument to java, e.g.,

```
java -classpath <some-path>:art.jar
        de.jaetzold.art.examples.HelloRobot
```

Any further explanation of the details of using JAVA, what a classpath is, and how to install and obtain JAVA or the JAVA Communications API on your computer is beyond the scope of this article.

If everything works as expected, executing the HelloRobot program will turn on a motor connected to the first port of the first robot interface found (for which a driver is available). Currently, the only supported robot interfaces are the RCX from the Lego Mindstorms Robotics Invention System (all versions, but only in conjunction with the serial IR-Tower) and the Intelligent Interface from FischerTechnik. Additionally a driver for a "virtual" robot interface is available which displays the values of two outputs and lets you edit the values of two inputs as shown in **Figure 2**. This driver can be used for testing purposes or when no real hardware interface is available.


**Figure 2:** Screenshot of the "virtual" robot interface's window.

Of course, it would also be possible to develop a driver for any other hardware. Describing such advanced details of ART is beyond the scope of this article and is described in [**4**].

### What `HelloRobot` does

The program is supposed to turn a motor on, so the first thing to do is to create a new `Motor` object and initialize a corresponding reference. This is done with the line

A more detailed description of the abstraction of actuators and sensors can be found in [4] and [5]. For example, the Sensor classes in ART can be chained one after another by using one sensor as a Port to which one or more other Sensor objects are connected. Although this is a powerful feature, it has been left out here for simplicity.

### Trusty with Subsumption

Subsumption is a concept for programming robots developed by Rodney A. Brooks [3]. It is based on the biological observation that complex behaviors sometimes result from many simpler behaviors that are like reflexes.

Programming Trusty with subsumption will at first result in a lot more code and thus it is not presented here. On the other hand, having many simple behaviors which can be connected again and again in many different combinations should be well suited for reuse of algorithms, because with ART it is possible to use the same algorithms for many different robots.

The Trusty example shown here is somewhat simplistic. The SimpleTrusty-algorithm does block the thread delivering the events for the time it avoids an obstacle. This is usually not desirable but avoiding that in this example would require a more complex programming.

The Trusty version which can be found in the examples from **http://www.jaetzold.de/art/** is programmed differently. It has a seperate JAVA interface -- just like in the DriveTrain example -- and includes different algorithms which are re-used in more sophisticated examples. One of these algorithms is implemented using subsumption and overcomes the problem of blocking the thread delivering the events.

### Trusty's Robot Implementation

Implementing the JAVA interface SimpleTrusty.Robot is not difficult. In the code shown here the actual methods of the JAVA interface, the ones that return the parameters, are omitted because they really do nothing else than returning the corresponding instance variables.

The implementation shown here is for one of my robots built from Lego. For other robots, also if they were built with FischerTechnik or any other robot hardware supported by ART, the only main differences would be the timing parameters for backing up and turning. Perhaps the sensors need some special configuration too or another implementation of DriveTrainSimpleAlgorithm.Robot or even DriveTrain is required. But at least for my FischerTechnik-version of Trusty, these changes are minimal and only require a seperate implementation of the respective "Robot" JAVA interfaces.

```
package de.jaetzold.art.examples;

import de.jaetzold.art.RobotInterface;
import de.jaetzold.art.RobotInterfaceStringDefinition;
import de.jaetzold.art.SensorPort;
import de.jaetzold.art.BooleanSensor;

public class SimpleTrustyRobot implements SimpleTrusty.Robot {
    protected DriveTrain driveTrain;
    protected BooleanSensor leftSensor;
```

```
        motor.connectWith(port);
```

From now on the `Motor` object can be used to manipulate the state of a real motor connected to the robot interface. The task of `HelloRobot` was to just turn a motor on:

```
        motor.on();
```

The rest of the code of `HelloRobot` just keeps the JAVA Virtual Machine running because otherwise the motor might instantly switch off again because the robot interface does not receive any more signals from the PC. Implementations of `RobotInterface` usually create their own threads to communicate with the interface harware, but they should be daemon threads that don't keep the virtual machine alive. This behavior of ART is different from the Abstract Window Toolkit of JAVA and allows programs to exit just by ending all non-daemon threads without using `System.exit()`.

## DriveTrain

In this section, a typical chassis for a mobile robot is programmed. A JAVA interface called `DriveTrain` describes what a chassis should be able to do:

```
package de.jaetzold.art.examples;

public interface DriveTrain {
    public void forward();
    public void backward();
    public void leftSpin();
    public void rightSpin();
    public void stop();
}
```

The methods need no further explanation because they should do what their names suggest. This java interface could be implemented for many different robots, although it might be difficult for some drive-designs to spin the robot in place.

One very simple robot design that is able to perform the moves demanded by `DriveTrain` has three wheels. Two wheels are driven by two motors, each driving one wheel, and a third one is the so called idler wheel. That wheel can not only turn freely around a horizontal axis, but also around a vertical one. This way the robot can easily move forward and backward by turning the two motor driven wheels in the same direction. If one wheel moves faster than the other, the robot moves along a curve. The robot spins in place if the motors turn the wheels in opposite directions. The idler wheel stabilizes the robot so that it doesn't fall over. Apart from that, it follows every move determined by the other two wheels.

Such robot designs are very common because they are easy to build and allow very flexible moves. They can be found in the construction manuals from the Lego RIS and the Mobile Robots set from FischerTechnik, but it is not difficult to build such a chassis by yourself.

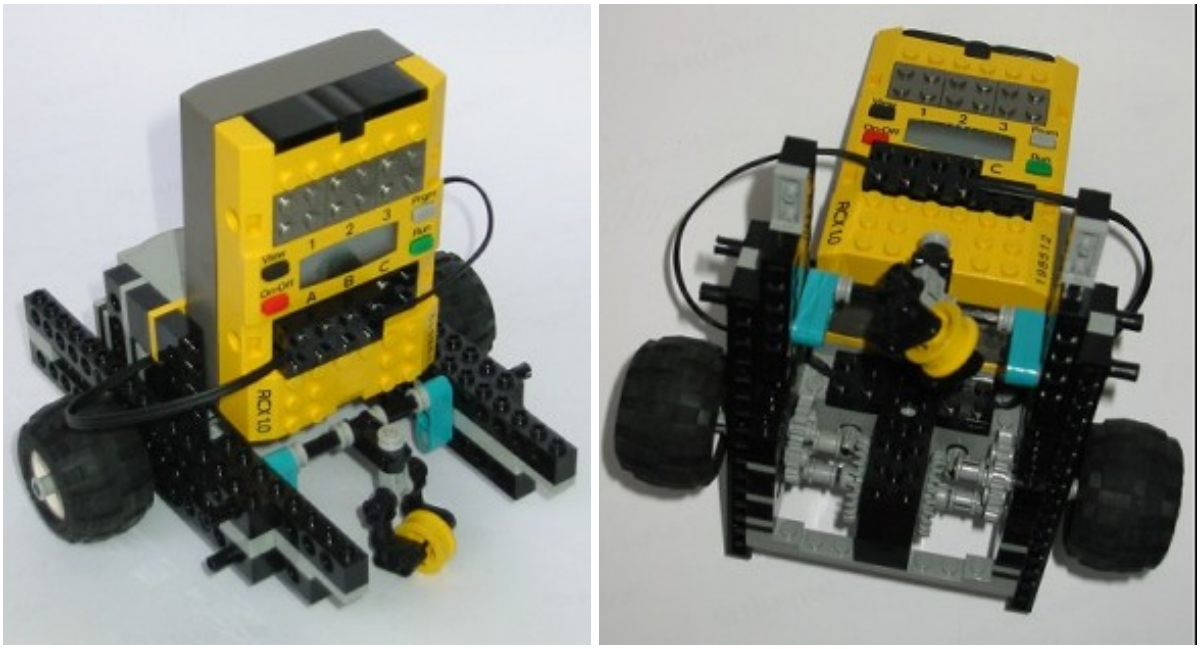

**Figure 3:** A simple chassis of a mobile robot.

The chassis shown in **Figure 3** has many gears between the motors and the wheels to slow it down. This is to work around the problem of the IR transmission between the RCX and the PC being relatively slow. With FischerTechnik, such workarounds are not necessary because the communication between the hardware interface and the PC is a lot faster. Nevertheless, I present the examples using Lego, because I believe the harware is much more widespread. This makes the current limitations of ART more obvious, but on the other hand it becomes possible to explain them. A description of both Lego and FischerTechnik can be found in [**4**].

The following code is an implementation of `DriveTrain` for a chassis as described above:

```
package de.jaetzold.art.examples;

import de.jaetzold.art.Motor;

public class DriveTrainSimpleAlgorithm implements DriveTrain {

    protected Motor left;
    protected Motor right;

    public DriveTrainSimpleAlgorithm(Robot robot) {
        this.left = robot.getLeftMotor();
        this.right = robot.getRightMotor();
        stop();
    }

    public static interface Robot {
        public Motor getLeftMotor();
        public Motor getRightMotor();
    }

    public void stop() {
        right.off();
        left.off();
```

```
    }

    public void forward() {
        right.forward();
        left.forward();

        right.on();
        left.on();
    }

    public void leftSpin() {
        right.forward();
        left.backward();

        right.on();
        left.on();
    }
    // ...
}
```

This implementation of the JAVA interface `DriveTrain` just needs a robot with two motors which meet the following requirements:

- If both motors are turning in the direction named "forward", the robot moves forward.
- If both motors are turning in the direction named "backward", the robot moves backward.
- If both motors are turning in opposite directions, the robot spins left if the left motor is the one turning "backward", and right if it is the right motor that turns "backward".
- The robot should stop moving if both motors are switched off.

The implementations of the methods `forward()`, `backward()`, `rightSpin()`, `leftSpin()` and `stop()` set the states of the motors accordingly. After setting the direction of the motors, they are also switched on because in ART whether a motor is on or off and the direction in which the motor turns are seen as independent of one another. This means if a motor is off and then receives a message setting its turning direction (e.g., `forward()`), nothing will happen with the real motor because it is still off.

Implementing the methods for `DriveTrain` is not complicated. The more interesting part of the code is at the beginning, where the variables `left` and `right` are initialized. The two motors are retrieved from another object which offers methods for this. This can be any object which implements the inner JAVA interface `DriveTrainSimpleAlgorithm.Robot`. This way, the responsibility for creating, connecting, and configuring the motors is kept seperate from the algorithm actually controlling the behavior of the motors. Any robot for which an implementation for `DriveTrainSimpleAlgorithm.Robot` can be created returning motors that meet the requirements specified above can be controlled by `DriveTrainSimpleAlgorithm`, and can in turn be used as a `DriveTrain`. This design-pattern is illustrated in **Figure 4**. Apart from having generalized classes to represent the parts of a robot, such as `Motor`, this is how the re-use of algorithms for different robots is mainly made possible.

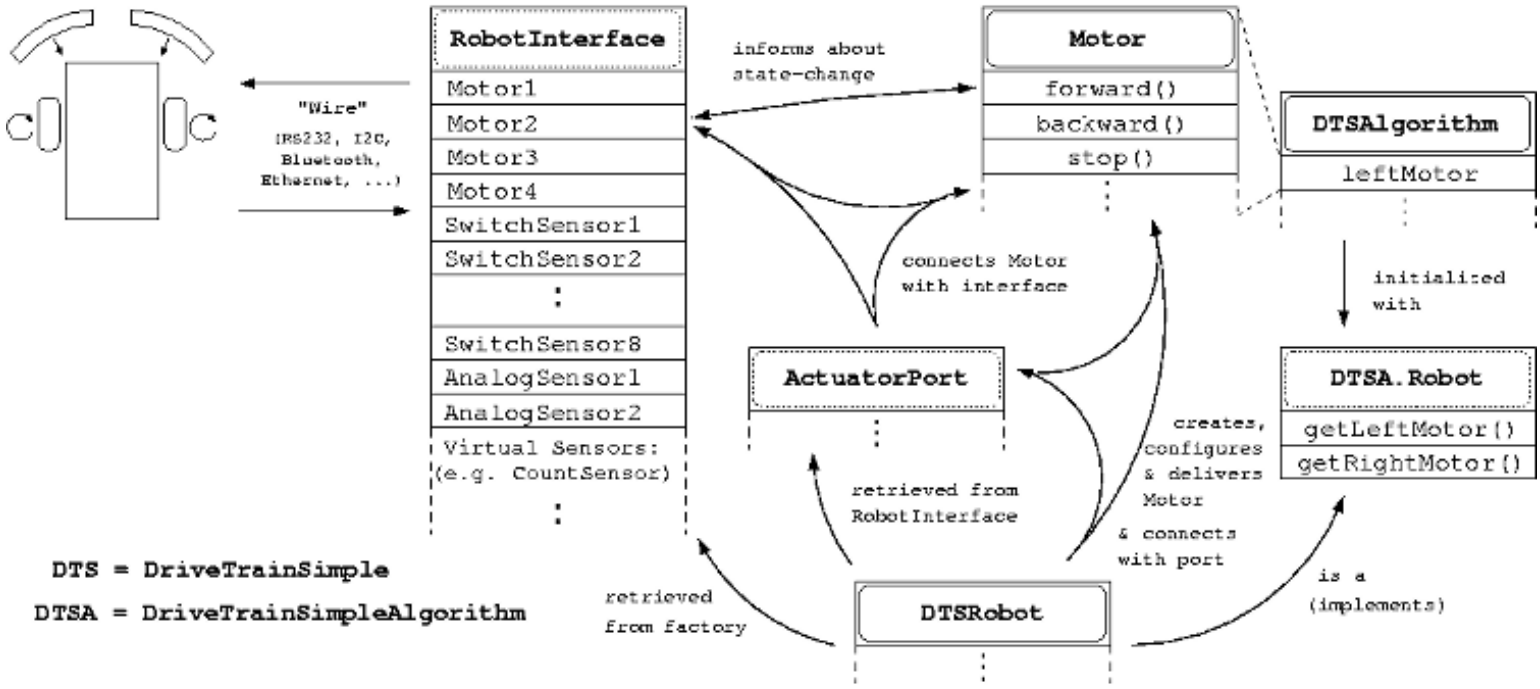


**Figure 4:** A possible design-pattern for programming robots with ART.

The implementation of `DriveTrainSimpleAlgorithm.Robot` shown here is not the shortest one imaginable. For example, it provides several different constructors: The main constructor already gets a `RobotInterface` as a parameter. It then checks that this `RobotInterface` provides enough `ActuatorPort` instances for connecting the two motors. You are already familiar with creating and connecting the motors; here we look at configuring the motors. Suppose the motors are installed so that they turn in the wrong direction by default. This might be the case because the cables connecting the motors with the RCX are so short that they fit only in a position reversing the polarity of the motors. ART can easily compensate for this: A `Motor` can be configured to interpret forward as backward and the other way round by sending it the message `setReversed(true)`:

```
package de.jaetzold.art.examples;

import de.jaetzold.art.Motor;
import de.jaetzold.art.ActuatorPort;
import de.jaetzold.art.RobotInterface;
import de.jaetzold.art.RobotInterfaceDefinition;
import de.jaetzold.art.RobotInterfaceStringDefinition;

public class DriveTrainSimpleRobot
                      implements DriveTrainSimpleAlgorithm.Robot
{
    protected Motor leftMotor;
    protected Motor rightMotor;

    public DriveTrainSimpleRobot(RobotInterface iface) {
        ActuatorPort[] ports = iface.getActuatorPorts();
        if(ports.length < 2) {
            throw new IllegalArgumentException(
                        "RobotInterface "
                        +iface
```

```
                            +" provides only "
                            +ports.length
                            +" Ports, minimum is 2.");
        }

        leftMotor = new Motor();
        leftMotor.connectWith(ports[0]);

        rightMotor = new Motor();
        rightMotor.connectWith(ports[1]);

        // the motors on the robot turn the other way round
        leftMotor.setReversed(true);
        rightMotor.setReversed(true);
    }

    public Motor getLeftMotor() {
        return leftMotor;
    }
    public Motor getRightMotor() {
        return rightMotor;
    }

    public DriveTrainSimpleRobot(RobotInterfaceDefinition definition) {
        this(new de.jaetzold.art.RobotInterfaceFactory()
                .getInterface(definition));
    }
    public DriveTrainSimpleRobot(String interfacePortName) {
        this(new RobotInterfaceStringDefinition(interfacePortName));
    }
    public DriveTrainSimpleRobot() {
        this("ALL");
    }
}
```

The other constructors of `DriveTrainSimpleRobot` are just for convenience. They allow you to use the class in a more naive way not needing to worry about creating a `RobotInterface`.

A `RobotInterfaceFactory` allows you to specify the types of `RobotInterface` you want it to create for you. This is done with a parameter of type `RobotInterfaceDefinition`. This type is nothing but an empty JAVA interface. That means virtually any class could be a `RobotInterfaceDefinition` as long as it declares to be one. Only the drivers feeling responsible for the given `definition`-object are asked to search for connected interfaces conforming to `definition`. This way it can be controlled, which drivers do their searching, and which not, and on which port they search. It might for example be the case that a driver's attempt to autodetect an interface interferes with some other hardware connected to this port, or that multiple interfaces are connected and you only want a specific one.

What type of `RobotInterfaceDefinition` is needed should be defined in the documentation of the driver for the desired robot interface. Each driver can have its own type. The drivers already available in ART all use the same type of `RobotInterfaceDefinition`, a `RobotInterfaceStringDefinition`. This

class encapsulates a `String` which defines a specific set of interfaces. Because strings are human-readable, this is an easy way to choose the interface one wants to receive from the factory, for example:

- `"MS"` -- the driver for Lego Mindstorms searches on all known (serial) ports for a connected RCX.
- `"MSCOM2"` -- the driver for Lego Mindstorms searches only on the port named `COM2` for an RCX. The name of the port depends on the implementation of the JAVA Communications API [**11**] used.
- `"FT/dev/ttyS0"` -- on the port named `/dev/ttyS0` the driver for the FischerTechnik Intelligent Interface searches for a connected interface.
- `"ALL1"` -- on the serial ports `COM1`, `/dev/ttyS0` and on the first portname delivered by the JAVA Communications API all drivers search for a connected interface. The string "ALL" should be accepted at least by all drivers accepting a `RobotInterfaceStringDefinition`, but maybe a driver does not use a serial port. How the rest of the string (the "1" in this example) is interpreted is totally up to the driver.

### Testing the algorithm

The classes presented so far still don't do anything by themselves -- there is no `main` method where execution could start. I encourage you to write one by yourself. It should not be difficult, just create a new `DriveTrainSimpleRobot` and use it as an argument when creating a new `DriveTrainSimpleAlgorithm`. Then test some of the methods of `DriveTrainSimpleAlgorithm` which are defined by the JAVA interface `DriveTrain`. Don't forget to give the robot some time to exhibit the appropriate behavior. In the first example ("Hello, Robot!"), you've already seen how you can wait for some time with a call like `Thread.sleep(1000)`.

If there is no real robot interface connected, the virtual driver is usually automatically used and the window shown in **Figure 2** opens. The `DriveTrain` example (along with many others) is also downloadable from **http://www.jaetzold.de/art/**. That version also includes a nice little `main` method which opens a window as shown in **Figure 5** on your desktop where you can press buttons to initiate calls to the methods of `DriveTrain` and view your robot's current state.

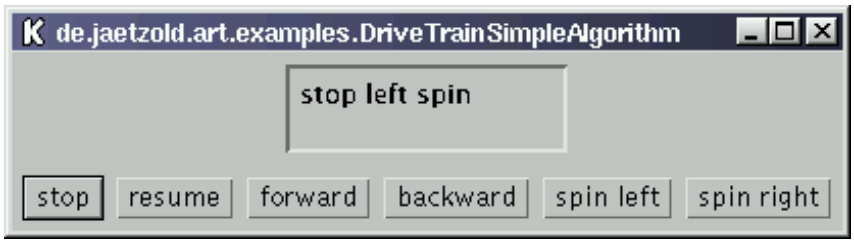


**Figure 5:** A window from a simple test application for a `DriveTrain`.

The previous example was not that complicated, but already four different classes and JAVA interfaces had to be created. The division was made to simplify re-use of the algorithm for different robots. However, it might sometimes be useful not to make this distinction with a JAVA interface for the algorithm and one or more implementations which in turn define a JAVA interface for the robot too (which also has to be implemented). This technique already produces a lot of classes for very simple problems, and if no re-use of the problem's solution in another context is imagineable, it might be overkill. On the other hand, further division of the implementation of `DriveTrainSimpleAlgorithm.Robot` can become necessary if, for example, configuring the motor objects is complex and parts of the implementation should be re-used for different robot designs or algorithms. This is used for the examples in [**4**] where

a second model built from FischerTechnik is used with the same algorithm.

In this article, the single implementation of `DriveTrain` is re-used in the next section, where sensors will be used so that the robot actually can demonstrate some reaction to its environment. This next example is done without a seperate JAVA interface for the algorithm to shorten the amount of code that has to be shown.

## Trusty

Up until now, you have only learned how to use motors with ART. A "real" robot, however, should also have sensor inputs to react to its environment. In this section simple touch sensors are used in conjunction with ART. There are also some classes in ART representing more complex sensors and actuators like a rotation sensor or a servo which will not be described here. If you are interested in them, try the javadoc documentation of ART or [**4**].

The robot discussed in this section is a very typical example for a mobile robot. Its name, Trusty, was originally used by Jonathan B. Knudsen in [**7**] for a robot that followed a line. The robot programmed here doesn't do that, but in a course at the University of Osnabrück, Germany, we used that name for a type of robot that moves around and avoids obstacles because of the common drive mechanism.

Apart from a chassis that allows Trusty to move around, it has a mechanism triggering a touch sensor whenever it bumps into something. In fact, there are even two such bumpers, one for each side, so that Trusty knows which side the obstacle is on, and therefore has better chances to swerve to the correct side. A schematic of this design is shown in **Figure 6**.

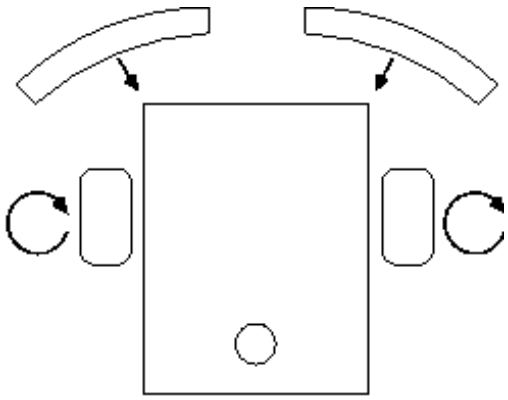

**Figure 6:** Schematics of the design of Trusty.

With Trusty's program, the distinction between the algorithm and a more platform-dependent `Robot` implementation is made again:

```
package de.jaetzold.art.examples;

import de.jaetzold.art.BooleanSensor;
import de.jaetzold.art.SensorListener;
import de.jaetzold.art.SensorEvent;

public class SimpleTrusty {
    protected DriveTrain chassis;
    protected int backupTime;
    protected int turnTime;
```

```
public void bumpedLeft() {
    chassis.backward();
    try {
        Thread.sleep(backupTime);
    } catch(InterruptedException ie) {
    }

    chassis.rightSpin();
    try {
        Thread.sleep(3*turnTime/4 +(int)(Math.random()*turnTime/2));
    } catch(InterruptedException ie) {
    }

    chassis.forward();
}

static interface Robot {
    public DriveTrain getDriveTrain();
    public BooleanSensor getLeftBumper();
    public BooleanSensor getRightBumper();
    public int getBackupTime();
    public int getTurnTime();
}

public SimpleTrusty(Robot robot) {
    chassis = robot.getDriveTrain();

    backupTime = robot.getBackupTime();
    turnTime = robot.getTurnTime();

    final BooleanSensor leftBumper = robot.getLeftBumper();
    leftBumper.addSensorListener(
        new SensorListener() {
            public void processEvent(SensorEvent se) {
                if(leftBumper.convertToBoolean(se.getValue())) {
                    bumpedLeft();
                }
            }
        }
    );

    final BooleanSensor rightBumper = robot.getRightBumper();
    rightBumper.addSensorListener(
        new SensorListener() {
            public void processEvent(SensorEvent se) {
                if(rightBumper.convertToBoolean(se.getValue())) {
                    bumpedRight();
                }
            }
        }
    );
```

```
        // start moving
        chassis.forward();
    }
}
```

When Trusty bumps into an obstacle, it backs up a bit, turns a few degrees to the side, and then moves forward again until it bumps into something again. It does that by letting the chassis move backward or spin and then wait for a few seconds (or milliseconds) until the chassis moves forward again. The method `bumpedRight()` for avoiding an obstacle on the right side, is omitted here for reasons of simplicity -- it is the same method as `bumpedLeft()` except that the chassis spins left and not right.

The `bumped`-methods are called whenever the corresponding touch sensor is pressed. Sensors that have two states, like pressed or not pressed as with a touch sensor, are represented in ART as instances of `BooleanSensor`.

Because `DriveTrain` only offers methods that make the robot move in one direction, but not for a specific distance or degrees (when turning), it is necessary to specify the time needed to back up "a bit" and turn "a few degrees". The algorithms should be platform independent: e.g., if a robot moves very fast, the algorithm has to wait for a shorter period of time until the robot backed up "a bit".

A `DriveTrain`, one `BooleanSensor` for the left and one for the right side, and the time needed to back up and turn are the necessary parameters for Trusty's algorithm `SimpleTrusty`. These parameters are collected as an object implementing the `Robot` JAVA interface inside the algorithm. The constructor of `SimpleTrusty` just retrieves these parameters from the given `robot`.

### Using sensors

Dealing with sensors is the new aspect here; therefore, it will be explained in more detail. The objects representing the sensors are ready to use when retrieved from the `robot`. The `robot` object already took care of creating a new instance of the right `Sensor` class, configuring it if necessary, and connecting it to a `Port` from a `RobotInterface`. This is just like with the `motor` objects from the `DriveTrain` example. The implementation of `SimpleTrusty.Robot` (where this all happens) is shown a bit later on.

Many algorithms using sensors can be programmed nicely in a way that is referred to as **event-driven**. A `Sensor` like the `BooleanSensor` can inform some "listeners" whenever a specific event occurs. This can be used just like the event mechanism in the AWT. Currently in ART only one sort of event is available. This event occurs every time the value of the sensor changes, and then all registered instances of `SensorListener` receive the message `processEvent` with a `SensorEvent` as parameter.

The listeners registered in this example call the method `bumpedLeft()` and `bumpedRight()` respectively whenever the value of the event is `true`. A `SensorEvent` in ART does not have booleans as values; therefore, the normal `double` value has to be converted. The default conversion is just like in C, only a value of 0 is `false` and any other value is `true`. But this conversion can be changed, and therefore it is better to let the `BooleanSensor`-instance itself convert the value. This is what the statement `leftBumper.convertToBoolean(se.getValue())` does.

A more detailed description of the abstraction of actuators and sensors can be found in [4] and [5]. For example, the Sensor classes in ART can be chained one after another by using one sensor as a Port to which one or more other Sensor objects are connected. Although this is a powerful feature, it has been left out here for simplicity.

### Trusty with Subsumption

Subsumption is a concept for programming robots developed by Rodney A. Brooks [3]. It is based on the biological observation that complex behaviors sometimes result from many simpler behaviors that are like reflexes.

Programming Trusty with subsumption will at first result in a lot more code and thus it is not presented here. On the other hand, having many simple behaviors which can be connected again and again in many different combinations should be well suited for reuse of algorithms, because with ART it is possible to use the same algorithms for many different robots.

The Trusty example shown here is somewhat simplistic. The SimpleTrusty-algorithm does block the thread delivering the events for the time it avoids an obstacle. This is usually not desirable but avoiding that in this example would require a more complex programming.

The Trusty version which can be found in the examples from **http://www.jaetzold.de/art/** is programmed differently. It has a seperate JAVA interface -- just like in the DriveTrain example -- and includes different algorithms which are re-used in more sophisticated examples. One of these algorithms is implemented using subsumption and overcomes the problem of blocking the thread delivering the events.

### Trusty's Robot Implementation

Implementing the JAVA interface SimpleTrusty.Robot is not difficult. In the code shown here the actual methods of the JAVA interface, the ones that return the parameters, are omitted because they really do nothing else than returning the corresponding instance variables.

The implementation shown here is for one of my robots built from Lego. For other robots, also if they were built with FischerTechnik or any other robot hardware supported by ART, the only main differences would be the timing parameters for backing up and turning. Perhaps the sensors need some special configuration too or another implementation of DriveTrainSimpleAlgorithm.Robot or even DriveTrain is required. But at least for my FischerTechnik-version of Trusty, these changes are minimal and only require a seperate implementation of the respective "Robot" JAVA interfaces.

```
package de.jaetzold.art.examples;

import de.jaetzold.art.RobotInterface;
import de.jaetzold.art.RobotInterfaceStringDefinition;
import de.jaetzold.art.SensorPort;
import de.jaetzold.art.BooleanSensor;

public class SimpleTrustyRobot implements SimpleTrusty.Robot {
    protected DriveTrain driveTrain;
    protected BooleanSensor leftSensor;
```

```
    protected BooleanSensor rightSensor;
    protected int backupTime;
    protected int turnTime;

    public SimpleTrustyRobot(DriveTrain driveTrain,
                             int backupTime,
                             int turnTime,
                             RobotInterface iface) {
        init(driveTrain, backupTime, turnTime, iface);
    }

    public SimpleTrustyRobot() {
        RobotInterface iface =
            new de.jaetzold.art.RobotInterfaceFactory()
                .getInterface(
                    new RobotInterfaceStringDefinition("ALL"));

        DriveTrain driveTrain =
                new DriveTrainSimpleAlgorithm(
                    new DriveTrainSimpleRobot(iface));

        init(driveTrain, 1500, 1000, iface);
    }

    private void init(  DriveTrain driveTrain,
                        int backupTime,
                        int turnTime,
                        RobotInterface iface) {

        this.driveTrain = driveTrain;
        this.backupTime = backupTime;
        this.turnTime = turnTime;

        SensorPort[] ports = iface.getSensorPorts();
        if(ports.length < 2) {
            throw new IllegalArgumentException(
                            "RobotInterface " +iface
                            +" provides only "
                            +ports.length
                            +" SensorPorts"
                            +", minimum is 2.");
        }

        leftSensor = new BooleanSensor();
        leftSensor.connectWith(ports[0]);

        rightSensor = new BooleanSensor();
        rightSensor.connectWith(ports[1]);
    }
}
```

One constructor already gets a `DriveTrain`, the timing parameters and a `RobotInterface` to which the sensors will be connected just like it has been done for a motor.

Another constructor defines default parameters. It creates a `DriveTrain` of type `DriveTrainSimpleAlgorithm` with a `DriveTrainSimpleAlgorithm.Robot` of type `DriveTrainSimpleRobot`. The `RobotInterface` used to create the `DriveTrainSimpleRobot` is then passed to `init()` so that both algorithms use the same robot interface. If you'd like, you could also use different instances of `RobotInterface` for `DriveTrainSimpleRobot` and for `SimpleTrustyRobot`. This is really easy and imagine how cool a robot would be that is actually built by combining several hardware interfaces that are totally different!

## Conclusion

For simple problems and where timing is not that important, ART is well suited for programming robots. It offers a degree of independence of the hardware that is seldom found in other programming systems for robots. For a system that allows to remotely control a hardware interface from a stationary PC with JAVA it is also relatively easy to use and understand, although the concepts may seem a bit strange sometimes, compared to other programming systems.

When programs controlling robots become reusable, it should be possible to put together complex behaviors by using preexisting simpler ones with subsumption. ART helps with this through making the behaviors reusable for many robots.

The division of the control program in an algorithm and a class describing and configuring the robot results in more code. But, the classes implementing one of these robot JAVA interfaces are all structured in the same way and mainly encapsulate some state. Therefore, it should be easily possible to describe such classes in a much simpler way or even click them together in a graphical editor, and let them be generated automatically from this information.

When thinking about an editor for clicking objects together, JAVA Beans [**10**] comes to mind. It would be nice if the objects from ART were JAVA Beans. Although some of the basic design patterns necessary for this already exist, a vital part is missing and its solution is really not obvious by now: How do we achieve persistence with objects representing parts of a robot, for example a motor?

## References


**1** Baum, Dave, and Gasperi, Michael, and Hempel, Ralph, and Villa, Luis *Extreme Mindstorms: An Advanced Guide to Lego Mindstorms.* Apress, 2000.

**2** Bollella, Greg, and Gosling, James, and Brosgol, Benjamin M., and Dibble, Peter, and Furr, Steve, and Hardin, David, and Turnbull, Mark. *The Realtime Specification for JAVA.* Addison Wesley, 2000.

**3** Brooks, Rodney A. *A Robust Layered Control System for a Mobile Robot.* September 1985. <**http://www.ai.mit.edu/people/brooks/papers/AIM-864.pdf**> (27 April 2002).

**4** Jätzold, Stephan. *Objekte, Threads und Events für Roboter. Ein Toolkit zur hardware-unabhängigen Robotersteuerung in JAVA.* February 2002. <**http://www-lehre.informatik.uni-osnabrueck.de/~fsjaetzo/art/art.pdf**> (27 April 2002).

**5** Jätzold, Stephan. "Programming robots with JAVA." *Slides from a talk at the Rochester Institute*

```
        // start moving
        chassis.forward();
    }
}
```

When Trusty bumps into an obstacle, it backs up a bit, turns a few degrees to the side, and then moves forward again until it bumps into something again. It does that by letting the chassis move backward or spin and then wait for a few seconds (or milliseconds) until the chassis moves forward again. The method `bumpedRight()` for avoiding an obstacle on the right side, is omitted here for reasons of simplicity -- it is the same method as `bumpedLeft()` except that the chassis spins left and not right.

The `bumped`-methods are called whenever the corresponding touch sensor is pressed. Sensors that have two states, like pressed or not pressed as with a touch sensor, are represented in ART as instances of `BooleanSensor`.

Because `DriveTrain` only offers methods that make the robot move in one direction, but not for a specific distance or degrees (when turning), it is necessary to specify the time needed to back up "a bit" and turn "a few degrees". The algorithms should be platform independent: e.g., if a robot moves very fast, the algorithm has to wait for a shorter period of time until the robot backed up "a bit".

A `DriveTrain`, one `BooleanSensor` for the left and one for the right side, and the time needed to back up and turn are the necessary parameters for Trusty's algorithm `SimpleTrusty`. These parameters are collected as an object implementing the `Robot` JAVA interface inside the algorithm. The constructor of `SimpleTrusty` just retrieves these parameters from the given `robot`.

### Using sensors

Dealing with sensors is the new aspect here; therefore, it will be explained in more detail. The objects representing the sensors are ready to use when retrieved from the `robot`. The `robot` object already took care of creating a new instance of the right `Sensor` class, configuring it if necessary, and connecting it to a `Port` from a `RobotInterface`. This is just like with the `motor` objects from the `DriveTrain` example. The implementation of `SimpleTrusty.Robot` (where this all happens) is shown a bit later on.

Many algorithms using sensors can be programmed nicely in a way that is referred to as **event-driven**. A `Sensor` like the `BooleanSensor` can inform some "listeners" whenever a specific event occurs. This can be used just like the event mechanism in the AWT. Currently in ART only one sort of event is available. This event occurs every time the value of the sensor changes, and then all registered instances of `SensorListener` receive the message `processEvent` with a `SensorEvent` as parameter.

The listeners registered in this example call the method `bumpedLeft()` and `bumpedRight()` respectively whenever the value of the event is `true`. A `SensorEvent` in ART does not have booleans as values; therefore, the normal `double` value has to be converted. The default conversion is just like in C, only a value of 0 is `false` and any other value is `true`. But this conversion can be changed, and therefore it is better to let the `BooleanSensor`-instance itself convert the value. This is what the statement `leftBumper.convertToBoolean(se.getValue())` does.