

Robotic Behaviors

What makes a robot a robot? It is the ability of the robot to independently accomplish complex tasks. Being able to independently do things is commonly thought of as a sign of intelligence. Developing intelligent machines, artificial intelligence, has long been a goal of computer scientists. Traditionally, artificial intelligence programs have been very large and extremely complex. That is, until Rodney Brooks of MIT's Artificial Intelligence Laboratory started thinking about insects.

Insects have very little computing capacity in their brains. Yet, a fly can find food, avoid predators and reproduce, all of which are complex behaviors. How does something with so little intelligence accomplish such complex tasks?

Basically, an insect simply reacts to its environment. The insect has a set of hard-wired behaviors. A particular environmental stimulus or event trips a behavior. The stimuli can be virtually anything. If something is moving quickly, it jumps. If the humidity goes up, it seeks a mate. If its sugar level gets too low, it seeks food. Etc. Note that these behaviors have different priority levels. Finding food is fairly high priority but avoiding a fly swatter is much higher. So, if the fly is trying to find food but it sees a fly swapper coming, it will immediately stop seeking food and flee.

The idea of insect behavior leads to a fundamentally simple way to control robots. Build a set of behaviors and associate each behavior with a trigger event. Assign each event-behavior pair a priority and then sit back and watch your robot go.

Our simple maze runner program has three separate behaviors, which we list at right. The default event starts when you press the run button. The associated behavior is to just go forward. When the robot runs into a wall, the default behavior is interrupted by one of the hit wall behaviors. After the hit wall behavior completes, the default behavior starts running again.

Default	Go forward
Hit wall on left	Back up, turn right
Hit wall on right	Back up, turn left

By creating a large set of event-behavior pairs and assigning them different priorities, a robot can perform very complex tasks.

Behaviors in LeJOS

A behavior is a Java object. You create a new behavior by creating a class that implements the `josx.robotics.Behavior` interface (UML at right). The details of how behaviors work in LeJOS are slightly different than the preceding description implies. Instead of directly calling an instance of a behavior when a stimulus event occurs, the behavior is asked if it wants to take control. If it does then the `suppress` method of the currently running behavior is called followed by the `action` method of the new behavior.

<code>josx.robotics.Behavior</code>
<none>
<code>boolean takeControl()</code> <code>void suppress()</code> <code>void action()</code>

Behaviors work in conjunction with `josx.robotics.Arbitrator`. *Arbitration* is the process of deciding which behavior should be running and that is what an Arbitrator does. The UML for Arbitrator is at right and you can see it is quite simple. When you create a new Arbitrator, you pass the constructor an array of Behaviors. The array is in order of priority with the highest priority behavior last in the list. After you create the new instance of Arbitrator, you call its `start` method to start arbitration. Note that the `start` method does not create a new thread for the arbitration to run on so the `start` method will never return.

<code>josx.robotics.Arbitrator</code>
<none>
<code>void start()</code>

The source code for both Behavior and Arbitrator are in `\lejos\classes\josx\robotics`. The code is straightforward and well commented. This is a good example of how you should code.

MazeRunner

MazeRunner is a program that we have looked at extensively. In the next few pages, we will analyze how to implement it using behaviors in Java. Note that there are three separate classes that will be detailed in the following sections. You can cut and paste each class into a separate file, named respectively, and compile them using lejos. Then you can download all three classes by using lejos on only MazeRunner, the other classes will be automatically downloaded with MazeRunner, to a Roverbot with double touch sensors.

MazeRunner comes in three separate files. The first is the MazeRunner class. It starts by importing all of the classes of `josx.robotics` so that it has access to `Behavior` and `Arbitrator`. Note that it does not declare any methods or fields other than `main` so it is simply a program.

`main` starts by creating three behaviors, two of class `HitWall` and one of class `GoForward`. Note that both of these classes implement the `Behavior` interface so that

```
import josx.robotics.*;

public class MazeRunner {

    static void main(String[] args) {
        Behavior turnRight = new HitWall(HitWall.TURN_RIGHT);
        Behavior turnLeft = new HitWall(HitWall.TURN_LEFT);
        Behavior move = new GoForward();
        Behavior[] ba = {move, turnRight, turnLeft};
        Arbitrator arb = new Arbitrator(ba);
        arb.start();
    }
}
```

an instance of either object can be assigned to a variable of type `Behavior` without an explicit cast. The `HitWall` constructor has one parameter. Constants from the class are used to specify which side the sensor is on.

The next statement creates an array of behaviors. The array does not specify a size. Instead, it initializes the array to the three behaviors that we created, `move`, `turnRight`, and `turnLeft`. The ability to put instances of different objects into a single array is called *polymorphism*. This is only allowed because these classes implement or extend the class of the array.

The next statement creates an `Arbitrator` instance with the array of behaviors that was just created. Note that order of parameters is important. The rightmost behavior, `turnLeft`, has the highest priority. The leftmost behavior, the lowest priority behavior, should be a default behavior, that is one that is always ready to run.

Finally, the last statement starts the `Arbitrator` running. This method never returns. It continually checks each behavior in priority order to see if it is ready to run. When a higher priority behavior is ready, it stops a lower priority behavior and starts the higher priority behavior. If no behavior is ready to run, it keeps looping until any behavior is ready.

GoForward is a very simple behavior. It starts by importing two packages, `josx.robotics` and `josx.platform.rcx`. The first package is imported to access the behavior interface while the second allows access to the static motor object.

The class definition states that this class implements the `Behavior` interface. Implementing an interface simply means that the class defines the methods listed in the interface. For `Behavior`, the required methods are `takeControl`, `suppress` and `action`. `GoForward` defines those three methods and nothing more, although it could have other methods or fields if needed.

`takeControl` is called by the arbitrator to ask if the behavior wants to run. Since this is the default behavior, it always is ready to run so it simply returns `true`.

`suppress` is called by the arbitrator for the currently running behavior when the arbitrator finds a higher priority behavior that wants to run. In this case, we stop the robot.

`action` is called by the arbitrator when it determines that a behavior should run. This is a very simple behavior; it turns on both motors in the forward direction and then waits 1000 milliseconds (1 second). Assuming no other behavior interrupts it, this allows the robot to go forward for 1 second before any other command is given to it. However, if a higher priority behavior says that it is ready to run, this behavior will be suppressed and the higher priority behavior will be activated.

The combination of `action` and `suppress` highlight the fact that arbitration runs on two separate threads. The main thread is constantly checking to see if higher priority behaviors are ready to run while a second thread is used to run the `suppress` and `action` methods of the behavior instances. The second thread is what actually controls the actions of the robot.

```
import josx.robotics.*;
import josx.platform.rcx.*;

class GoForward implements Behavior {

    public boolean takeControl() {
        return true;
    }

    public void suppress() {
        Motor.A.stop();
        Motor.C.stop();
    }

    public void action() {
        Motor.A.forward();
        Motor.C.forward();
        try {
            Thread.sleep(1000);
        } catch (Exception e) {
        }
    }
}
```

HitWall imports the same packages and has the same implements clause as GoForward. However, it is a more complex behavior.

The two constants, TURN_LEFT and TURN_RIGHT, allow this class to be used for the sensor on either the right or left side of the robot. When the constructor is called, one of the constants is specified to set up the behavior. Since the parameter name for the constructor is the same as the field name, the this keyword is used so that we can save the value of the parameter in the instance.

takeControl is more complex since this behavior will not be used as a default behavior. That is, the only time that we want this behavior to run is when the appropriate sensor is pressed. Here when the arbitrator asks, we return the value of the appropriate sensor using a conditional operator. In this case, if this was instantiated as a turn left behavior we check the value of the sensor on the right and return it. If the robot is in contact with the wall on the right, the value of the sensor is true. The arbitrator then calls our action method.

suppress is the same as in GoForward and, in fact, this is a pretty generic suppression method.

action is somewhat more complex. First, it turns both motors on in reverse for 500 milliseconds (.5 seconds). Then it spins the robot by having the motors turn in opposite directions with the direction of the spin determined by the parameter specified when the behavior was constructed. The robot spins, again for .5 seconds. Note that both the amount of time for backing up and the time for spinning need to be calibrated for the individual robot. Note also that this behavior is exactly the same algorithm that we implemented in RIS code.

```
import josx.robotics.*;
import josx.platform.rcx.*;

class HitWall implements Behavior {

    private int direction = TURN_LEFT;
    static final int TURN_LEFT = 1;
    static final int TURN_RIGHT = 2;

    HitWall(int direction) {
        this.direction = direction;
    }

    public boolean takeControl() {
        return (direction == TURN_LEFT) ?
            Sensor.S3.readBooleanValue() :
            Sensor.S1.readBooleanValue();
    }

    public void suppress() {
        Motor.A.stop();
        Motor.C.stop();
    }

    public void action() {
        Motor.A.backward();
        Motor.C.backward();
        try {
            Thread.sleep(500);
        } catch (Exception e) {
        }
        if (direction == TURN_LEFT) {
            Motor.A.backward();
            Motor.C.forward();
        } else if (direction == TURN_RIGHT) {
            Motor.A.forward();
            Motor.C.backward();
        }
        try {
            Thread.sleep(500);
        } catch (Exception e) {
        }
    }
}
```