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, these behaviors have different priority levels. Finding food is a 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 two separate behaviors, which we list at right. The default behavior starts when you press the run button and it is to just go forward. When the robot runs into a wall, the default behavior is interrupted by the hit wall behavior. After the hit wall behavior completes, the default behavior starts running again.

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

<table>
<thead>
<tr>
<th>Default</th>
<th>Go forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit wall</td>
<td>Back up, turn left or right</td>
</tr>
</tbody>
</table>
**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.

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 is 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.

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. Add them to your RCX project to ensure that they will be compiled using the LeJOS libraries. Note that two of the classes extend ZTRBot, which you should have already created. If you have not created ZTRBot, created it in a package other than MyRobots or haven’t added all of the methods that are used, you will get compiler errors. You must have a working ZTRBot with all the required methods to build MazeRunner.

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 two behaviors, one of class HitWall and one of class GoForward. Note that both of these classes implement the Behavior interface so that an instance of either object can be assigned to a variable of type Behavior.

The next statement creates an array of behaviors. The array does not specify a size. Instead, it initializes the array to the two behaviors that we created, move and wall. The ability to put instances of different objects into a single array is called polymorphism. This is only allowed because these classes implement the Behavior interface.

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, wall, has the highest priority. The leftmost behavior, move, is the lowest priority behavior and should be a default behavior, i.e. a behavior 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 with a package statement so that it belongs to our MyRobot set of classes. It then imports josx.robotics.Behavior so that Behavior is defined. Next, the class is defined as extending ZTRBot and implementing Behavior. Extending ZTRBot gives us all the movement methods for our robot. 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.

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.

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.

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.
HitWall belongs to the same package as the other classes. It imports Behavior, just as GoForward does, and also imports Sensor so that it can read the values of the sensors and so interact with the environment. It extends ZTRBot so that it has access to the methods used to move the robot and implements Behavior just like GoForward.

takeControl is more complex than in GoForward 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 a sensor is pressed. Here when the arbitrator asks, we return true if sensor 1 or sensor 3, or both, is pressed. The arbitrator then calls our action method.

action starts by asking if sensor 1 was pressed. If so, it backs up and turns right (this is away from the wall if the robot is correctly constructed). If not then it assumes that sensor 3 was pressed and so backs up and turns left (again, away from the wall). We use the “if … else” statement since takeControl returning true guarantees at least one sensor is pressed and if sensor 1 is not pressed then sensor 3 must be pressed. We don’t include any statement to make the robot start going forward again. That is the role of the default behavior, GoForward, which starts running again as soon as this behavior, HitWall, completes its action method.

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