LEJOS
Java for LEGO Mindstorms

« Previous • TOC • Next »
Home Page > Behavior programming

Behavior programming

Programming Behavior with leJOS NXJ

When most people start programming a robot, they think of the program flow as a series of if-thens, which is reminiscent of structured programming (Figure 1). This type of programming is very easy to get started in and hardly requires any thought or design beforehand. A programmer can just sit at the computer and start typing (although a little thought before the typing may avoid a lot of grief later.)

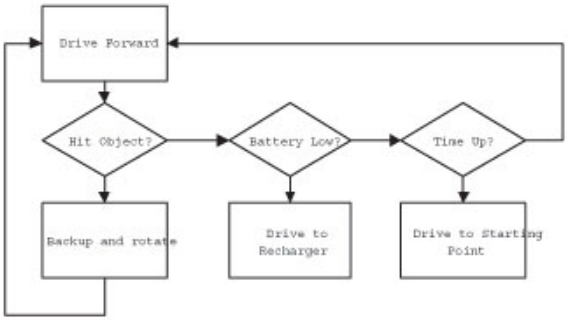

Fig 1 Structured programming visualized.

The problem is, the code ends up as spaghetti code; all tangled up and difficult to expand.
The behavior control model, in contrast, requires a little more planning before coding begins, but the payoff is that each behavior is nicely encapsulated within an easy to understand structure. This will theoretically make your code easier to understand by other programmers familiar with the behavior control model, but more importantly it becomes very easy to add or remove specific behaviors from the overall structure, without negative repercussions to the rest of the code. It also makes it possible to test and debug each behavior by itself.

The concepts of Behavior Programming as implemented in leJOS NXJ are very simple:

- Only one behavior can be active and in control of the robot at any time.
- Each behavior has a fixed priority.
- Each behavior can determine if it should take control.
- The active behavior has higher priority than any other behavior that should take control.

The Behavior API

The Behavior API is is composed of only one interface and one class. The Behavior interface defines the individual behavior classes. The Behavior interface is very general and defines three public methods. Each task that the robot must perform can be defined in its own class. It works quite well because, even though the individual implementations of a behavior vary widely, they are all treated alike. Once all the behaviors are created, they are given to an Arbitrator to regulate which behavior should be activated at any time. The Arbitrator class and the Behavior interface are located the lejos.subsumption package. The API for the Behavior interface is as follows.

lejos.subsumption.Behavior

- boolean takeControl()

  Returns a boolean value to indicate if this behavior should become active. For example, if a touch sensor indicates the robot has bumped into an object, this method should return true. This method should return quickly, not perform a long calculation.

- void action()

The code in this method begins performing its task when the behavior becomes active. For example, if takeControl() detects the robot has collided with an object, the action() code could make the robot back up and turn away from the object.

A behavior is active as long as its action() method is running, so the action() method should exit when its task is complete. Also, the action() method should exit promptly when suppress() is called. When it exits, it should leave the robot in a safe state for the next behavior.

- void suppress()

  The code in the suppress() method should immediately terminate the code running in the action() method. It also should exit quickly.

As you can see, the three methods in the Behavior interface are quite simple. If a robot has three discrete behaviors, then the programmer will need to create three classes, with each class implementing the Behavior interface. Once these classes are complete, your code should hand the behavior objects off to the Arbitrator to deal with.

lejos.subsumption.Arbitrator

The constructor is:

- public Arbitrator(Behavior[] behaviors, boolean returnWhenInactive)

  Creates an Arbitrator object that regulates when each of the behaviors will become active. Parameter: behaviors. The priority of each behavior is its index in the array. So behaviors[0] has the lowest priority.

  Parameter: boolean returnWhenInacative; If **true**, the program exits when there is no behavior that wants to take control. Otherwise, the program runs until shut down by pressing the Enter and Escape buttons.

  Parameter: an array of Behaviors

Public Methods:

- public void start()

  Starts the arbitration system.

The Arbitrator class is even easier to understand than Behavior. When an Arbitrator object is instantiated, it is given an array of Behavior objects. Once it has these, the start() method is called and it begins arbitrating; deciding which behavior will become active. The Arbitrator calls the takeControl() method on each Behavior object, starting with the object with the highest index number in the array. It works its way down through the array, (in decreasing priority order) till it finds a behavior that wants to take control. If the priority index of this behavior is greater than that of the current active behavior, the active behavior is suppressed. The action method is then called on the behavior of this index. As a result, if several behaviors want to take control, then only the highest priority behavior will become active. (Figure 2).

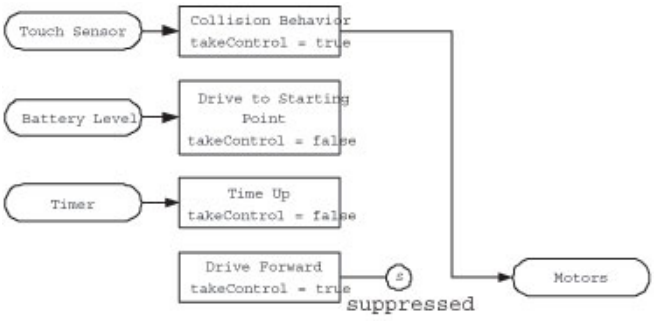

Figure 2: Higher level behaviors suppress lower level behaviors.

Coding Behaviors

For reliable performance of the behavior control system, it is essential that the action() method terminates promptly when suppress() is called. One way to guarantee this is to define a boolean flag *suppressed* in each behavior. This variable is set to *true* by the suppress() method and tested in every loop of the action() method. Of course, the first thing an action() method must do is set this flag to *false*. The action() method might be quite complex, such as a separate thread for line tracking or wall following, but it must be coded to ensure prompt exit from action() when necessary. This is the **recommended design pattern**.

Now that we are familiar with the Behavior API under leJOS, let's look at a simple example using three behaviors. For this example, we will program some behavior for a simple robot with differential steering. This robot will drive forward as its primary low-level behavior. This activity continues unless the robot hits an object, then a high priority behavior will become active to back the robot up and turn it 90 degrees. There will also be a third behavior which we will insert into the program after the first two have been completed. Let's start with the first behavior.

As we saw in the Behavior interface, we must implement the methods action(), suppress(), and takeControl(). The behavior for driving forward will take place in the action() method. It simply needs to make motors A and C rotate forward and it exits when the motors are no longer moving, or suppress is called. This behavior remains active as long as the motors are turning. The method code is:

```
public void action() {
   suppressed = false;
   Motor.A.forward();
   Motor.C.forward();
   while( !suppressed )
      Thread.yield();
   Motor.A.stop(); // clean up
   Motor.C.stop();
}
```

That was easy enough! Now the standard suppress() method will stop this action when it is called:

```
public void suppress() {
   suppressed = true;
}
```

So far, so good. Now we need to implement a method to tell Arbitrator when this behavior should become active. As we outlined earlier, this robot will drive forward always, unless something else suppresses it, so this behavior should always want to take control (it's a bit of a control freak). The takeControl() method should return true, no matter what is happening. This may seem counter-intuitive, but rest assured that higher level behaviors will be able to cut in on this behavior when the need arises. The method appears as follows:

```
public boolean takeControl() {
   return true;
}
```

That's all it takes to define our first behavior to drive the robot forward. The complete code listing for this class is as follows:

```
import lejos.nxt.*;
import lejos.robotics.subsumption.*;

public class DriveForward  implements Behavior {
   private boolean suppressed = false;

   public boolean takeControl() {
      return true;
   }
```

```
   public void suppress() {
      suppressed = true;
   }

   public void action() {
     suppressed = false;
     Motor.A.forward();
     Motor.C.forward();
     while( !suppressed )
        Thread.yield();
     Motor.A.stop(); // clean up
     Motor.C.stop();
   }
}
```

The second behavior is a little more complicated than the first, but still very similar. The main action of this behavior is to reverse and turn when the robot strikes an object or detects one close by. In this example, we would like the behavior to take control only when the touch sensor strikes an object, or the ultrasonic sensor gets an echo from a close object. Here is the takeControl() method that does it:

```
public boolean takeControl() {
  return touch.isPressed() || sonar.getDistance() < 25;
}
```

This assumes that a TouchSensor object has been created in an instance variable called *touch* and a new UltraSonicSensor object has been assigned to the variable *sonar*.

For the action, we want the robot to back up and rotate when it detects an object, so we will define the action() method as follows:

```
public void action()
{
    // Back up and turn
    suppressed = false;
    Motor.A.rotate(-180, true);
    Motor.C.rotate(-360, true);

    while(Motor.C.isRotating() && !suppressed)
        Thread.yield();  // wait till turn is complete or suppressed is called

    Motor.A.stop();
    Motor.C.stop();
}
```

We use the standard suppress() method here too.

```
public void suppress {
    suppressed = true;
}
```

The complete listing for this behavior is as follows:

```
import lejos.nxt.*;
import lejos.robotics.subsumption.*;

public class HitWall implements Behavior {
    private TouchSensor touch;
    private UltrasonicSensor sonar;
    private boolean suppressed = false;

    public HitWall(SensorPort port )
    {
       sonar = new UltrasonicSensor( port );
    }

    public boolean takeControl() {
```

```
        return touch.isPressed() || sonar.getDistance() < 25;
    }

    public void suppress() {
        suppressed = true;
    }

    public void action() {
        suppressed = false;
        Motor.A.rotate(-180, true);
        Motor.C.rotate(-360, true);

        while( Motor.C.isMoving() && !suppressed )
          Thread.yield();

        Motor.A.stop();
        Motor.C.stop();
    }
}
```

We now have our two behaviors defined, and it's a simple matter to make a class with a main() method to get things started. All we need to do is create an array of our behavior objects, and instantiate and start the Arbitrator as shown in the following code listing:

```
import lejos.nxt.SensorPort;
import lejos.robotics.subsumption.Arbitrator;
import lejos.robotics.subsumption.Behavior;

public class BumperCar {
   public static void main(String [] args) {
      Behavior b1 = new DriveForward();
      Behavior b2 = new HitWall(SensorPort.S2);
      Behavior [] bArray = {b1, b2};
      Arbitrator arby = new Arbitrator(bArray);
      arby.start();
   }
}
```

The above code is fairly easy to understand. The first two lines in the main() method create instances of our behaviors. The third line places them into an array, with the lowest priority behavior taking the lowest array index. The fourth line creates the Arbitrator, and the fifth line starts the Arbitration process. When this program is started the robot will scurry forwards until it bangs into an object, then it will retreat, rotate, and continue with its forward movement until the power is shut off.

This seems like a lot of extra work for two simple behaviors, but now let's see how easy it is to insert a third behavior without altering any code in the other classes. This is the part that makes behavior control systems very appealing for robotics programming. Our third behavior could be just about anything. We'll have this new behavior monitor the battery level and play a tune when it dips below a certain level. Examine the completed Behavior:

```
import lejos.nxt.Battery;
import lejos.nxt.Sound;
import lejos.robotics.subsumption.Behavior;

public class BatteryLow implements Behavior {
   private float LOW_LEVEL;
   private boolean suppressed = false;
   private static final short [] note = {
      2349,115, 0,5, 1760,165, 0,35, 1760,28, 0,13, 1976,23,
      0,18, 1760,18, 0,23, 1568,15, 0,25, 1480,103, 0,18,
      1175,180, 0,20, 1760,18, 0,23, 1976,20, 0,20, 1760,15,
      0,25, 1568,15, 0,25, 2217,98, 0,23, 1760,88, 0,33, 1760,
      75, 0,5, 1760,20, 0,20, 1760,20, 0,20, 1976,18, 0,23,
      1760,18, 0,23, 2217,225, 0,15, 2217,218};

   public BatteryLow(float volts) {
      LOW_LEVEL = volts;
   }
```

```
    public boolean takeControl() {
       float voltLevel = Battery.getVoltage();
       System.out.println("Voltage " + voltLevel);

       return voltLevel < LOW_LEVEL;
    }

    public void suppress() {
       suppressed = true;
    }

    public void action() {
       suppressed = false;
       play();
       System.exit(0);
    }

    public void play() {
       for(int i=0; i<note.length; i+=2) {
          final short w = note[i+1];
          Sound.playTone(note[i], w);
          Sound.pause(w*10);
          if (suppressed)
                  return; // exit this method if suppress is called
       }
    }
}
```

The complete tune is stored in the note array at line 6 and the method to play the notes is at line 30. This behavior will take control only if the current battery level is less the voltage specified in the constructor. The takeControl() method looks a little inflated, and that's because it also displays the battery charge to the LCD display. The action() method is comparatively easy. Action makes a bunch of noise, then exits the program. Since this behavior stops the program, we might get away without a a suppress() method. But just in case we detect a wall, while playing the tune, we use one.

To insert this behavior into our scheme is a trivial task. We simply alter the code of our main class as follows:

```
import lejos.nxt.SensorPort;
import lejos.robotics.subsumption.Arbitrator;
import lejos.robotics.subsumption.Behavior;

public class BumperCar {
   public static void main(String [] args) {
      Behavior b1 = new DriveForward();
      Behavior b2 = new BatteryLow(6.5f);
      Behavior b3 = new HitWall(SensorPort.S2);
      Behavior [] bArray = {b1, b2, b3};
      Arbitrator arby = new Arbitrator(bArray);
      arby.start();
   }
}
```

Note: The voltage level of the NXT at rest is different from the voltage when in action. The voltage level at rest might be 7.8 V, but when motors are activated they naturally cause a drop in the voltage reading. Make sure the voltage threshold used in the BatteryLow constructor is low enough.

This example beautifully demonstrates the real benefit of behavior control coding. Inserting a new behavior, no matter what the rest of the code looks like, is simple. The reason for this is grounded in object oriented design; each behavior is a self contained, independent object.

TIP: When creating a behavior control system, it is best to program each behavior one at a time and test them individually. If you code all the behaviors and then upload them all at once to the NXT brick, there is a good chance a bug will exist somewhere in the behaviors, making it difficult to locate. By programming and testing them one at a time it makes it easier to identify where the problem was introduced.

Back to Top

Notes on the recommended design pattern

In order to understand why we recommend this design pattern, it is useful to dig a little deeper into the inner workings of the arbitrator. The Arbitrator contains a monitor thread that cycles through each of the behaviors, checking the takeControl() method to see if the behavior should become active. It starts with behavior of largest index (because they are are stored in increasing priority order) and works down the array. As soon as it comes across a behavior that should take control, this is the highest priority behavior that wants control at the moment. If this behavior has higher priority than the active behavior, the monitor thread executes suppress() on the active behavior, and then starts checking each behavior from the top again. The main thread of the Arbitrator is very simple. It just calls the action() method on the highest priority behavior, (as determined by the Monitor thread) and that behavior becomes the active. When action() exits, either because its task is complete or because the Monitor thread has called suppress in it, that behavior is no longer active, and the loop continues, calling action() on the behavior that is now has the highest priority as determined by the Monitor thread. It is possible that the same behavior repeatedly becomes active.

It would be nice if all behaviors were as simple as the examples given above, but in more complex coding there are some unexpected results that can sometimes be introduced. If it is necessary to have a behavior that uses its own Thread, for example, can sometimes be difficult to halt it from the suppress() method, which can lead to two different threads fighting over the same resources - often the same motor! But in this pattern, the suppress() method does not try to halt anything. It is the responsibility of the action() method to keep testing the `suppressed` variable, exit as soon as it become true, and leave the robot is a safe state for the next behavior.

Another advantage of this design pattern is that each behavior is coded in the same way without making any assumptions about its priority. You can then change the priority order of your behaviors or reuse them in other applications without reprogramming any of them.

Note: If you would like to remove any mystery about what goes on in the Arbitrator class, take a look at the source code located in src/classes/lejos/subsumption/Arbitrator.java.

Summary

Behavior coding is predominantly used for autonomous robots - robots that work independently, on their own free will. A robot arm controlled by a human would likely not use behavior programming, although it would be possible. For example, a robot arm with four joystick movements could have a behavior for each direction of movement. But as you may recall, behaviors are ordered with the highest order taking precedence over lower order behaviors. Who is to say that pushing left on the joystick would take precedence over pushing up? In other words, behavior control in anything other than autonomous robots is largely overkill.

So why use the Behavior API? The best reason is because in programming we strive to create the simplest, most powerful solution possible, even if it takes slightly more time. The importance of reusable, maintainable code has been demonstrated repeatedly in the workplace, especially on projects involving more than one person. If you leave your code and come back to it several months later, the things that looked so obvious suddenly don't anymore. With behavior control, you can add and remove behaviors without even looking at the rest of the code, even if there are 10 or more behaviors in the program. Another big plus of behavior control is programmers can exchange behaviors with each other easily, which fosters code reusability. Hundreds of interesting, generic behaviors could be uploaded to websites, and you could simply pick the behaviors you want to add to your robot (assuming your robot is the correct type of robot). This reusability of code can be taken forward even more by using standard leJOS NXJ classes such as the Navigation API.

Back to Top

« Previous • TOC • Next »