Teaching Introductory Java through LEGO MINDSTORMS Models

David J. Barnes
The Computing Laboratory
The University
Canterbury, Kent. CT2 7NF
United Kingdom
d.j.barnes@ukc.ac.uk

Abstract

Innovative teachers are continually looking for creative ideas, both to get their ideas across and to hold the interest of their students. One of the latest trends is the use of LEGO® MINDSTORMS™ kits [9] in various computing courses. These kits allow a wide variety of physical models to be built, some of which may be programmed via the RCX™ processor integrated into them. Using its standard firmware, the RCX device may be programmed through several different specialist languages. However, the additional availability of bytecode-compatible replacement firmware for the RCX makes the use of Java™ as the programming language for it a particularly attractive approach. In this paper, we explore some of the issues associated with choosing to program MINDSTORMS models using Java within the context of an introductory programming course. In particular, we consider the impact on the material that is taught, and the use of an appropriate API to support an objects-early programming style.

1 The RCX Processor

The RCX is a programmable processor housed in an oversized LEGO brick. This allows the processor to be an integral part of any model that is built with the MINDSTORMS kits. On the outside of the processor's brick are three input ports (labeled 1, 2, and 3), three output ports (labeled A, B, and C), an input-output infrared device, a single-line LCD, a speaker, and four buttons (one of which is the on-off switch). In size, the input and output ports are compatible with standard LEGO bricks but they also have electrical contacts. These are designed to attach to similar contacts housed in special purpose input and output devices. A number of input-output devices are available as standard, such as motors, touch sensors, light sensors, rotation sensors, etc. The configuration of the RCX and the range of available devices make it possible to create a wide variety of programmed LEGO models. However, in this paper, our main interest is in the programming aspects of modeling and we do not intend to discuss any particular model in detail. The reader interested in exploring construction aspects is referred to references such as [3,5,7,10] for some of the ever-growing range of highly imaginative designs.

SIGCSE'02, February 27-March 3, 2002, Covington, Kentucky, USA.

2 RCX Programming Environments

With a standard MINDSTORMS kit, programs are created in RCX Code using a PC-based graphical programming environment. These programs are then uploaded via the environment into the RCX over an infrared link. A popular alternative to using RCX Code is Dave Baum's NQC [3] – a textual C-like language with greater versatility than RCX Code. In addition, a number of people have pieced together the internal details of the MINDSTORMS kits [11], enabling unofficial projects to create alternative firmware for the RCX, such as legOs [12] and pbForth [6]. Our particular interest is in the leJOS project [15], which has created replacement firmware that is compatible with the standard bytecode produced by Java compilers. This opens up the possibility of programming RCX-based models using Java.

3 The Appeal of Computer-Controlled Models

For a long time, educators have found the use of computer-controlled models to be an aid in the teaching of introductory programming. Turtle [13] is one of the best-known examples, and Karel the Robot [14] is one that started life as a paper-and-pencil approach but was accompanied by optional simulation. Karel continues to be used to teach programming [4], while other textbooks use related ideas such as modeling and navigating a ship [1].

The availability of kits such as MINDSTORMS makes it relatively easy to build and create interesting and imaginative physical models for students to program and control. One of the biggest advantages of such kits – for

both students and teachers – is surely that they need to know almost nothing about hardware; yet they can still create quite sophisticated working models.

One reason why physical models are attractive is that they provide tangible feedback to students on the workings of the programs [16]. Furthermore, the physical constraints of the coding and debugging cycle mean that good design and planning have to have a high priority in students' thinking [17]. In addition, for some students, controlling a physical model is likely to have much more appeal than controlling a graphical representation of what is, after all, meant to be a simulation of a physical model.

In the remainder of this paper, we consider some of the impacts that using RCX-based models in an introductory programming course could have on both the Java material that is taught and the object-oriented modeling aspects. For the purpose of discussion, we focus on the leJOS environment and its associated Java API [15].

4 An Illustrative Programming Example

Figure 1 illustrates a simple program that uses the leJOS API to drive a wheeled model.

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

/**
 * A model with left and right motors,
 * plus a single touch sensor.
 */
public class OneBumper {
   private Motor left = Motor.A,
                 right = Motor.C;
   private Sensor bumper = Sensor.S1;

   /**
    * Run forward until an obstacle is
    * hit. Then reverse.
    */
   public void run(){
      bumper.setTypeAndMode(
        SensorConstants.SENSOR_TYPE_TOUCH,
        SensorConstants.SENSOR_MODE_BOOL);
      left.setPower(3);
      right.setPower(3);
      left.forward();
      right.forward();
      // Wait until we hit a wall.
      while(bumper.readValue() == 0){
         // Do nothing.
      }
      // Back away from the wall.
      left.backward();
      right.backward();
   }
}
```

Figure 1: A model with two motors and one touch sensor

The program causes the model to move forward until its touch sensor is triggered, at which point the motors are reversed. For the sake of simplicity, we have not attempted to show any further interaction with the model once it is moving backwards. Polling is used to determine when the front bumper hits a wall; the value of the sensor is read repeatedly until its value changes from zero to one, which indicates a collision.

In the next section we discuss some of the issues that arise from this example in the context of an introductory Java course.

5 Introductory Programming Issues

Programs based around the example in Figure 1 might be used relatively early in a Java course to illustrate introductory features of object-oriented programming, such as statement sequences, method calling, parameter passing, and multiple instances of a class. However, there are some aspects that it does not illustrate particularly well. Consider the portion involving the while loop, for instance.

- This is not a good exemplar of a loop, as it has an empty body. Normally one would teach that the body of a loop contains statements that will eventually cause the loop's condition to become false.
- Polling is not always the best way to interact with input devices, particularly when there is more than one to deal with.
- A touch-sensor is typically used as an on-off device; we are only interested in whether it has been pressed or released. However, the `Sensor` class returns its state as an integer value. As Java supports a Boolean type, it would be preferable pedagogically to use genuine Boolean values for such sensors.

Figure 2 presents an alternative version of this example that seeks to address the first two of these points.

Instead of using polling to detect changes in the sensor, we have attached an event listener to it. This is a more natural way to monitor sensors, and scales well with multiple sensors. A negative aspect of using listeners, however, is that the programming concepts involved are significantly more complex than using polling. The implementation in Figure 2 uses both interfaces and anonymous inner classes, for instance, neither of which would normally be considered to be easy introductory-level concepts.

Another issue that may be difficult for introductory students is the link between the control flow within a driving program and the sequence of actions that a physical model goes through. The contrasting styles of polling and event listening present some conceptual differences in this respect. In Figure 1, the presence of the while loop tends to suggest – albeit fallaciously – that the continuous execution of the loop corresponds to the continuous forward movement of the model. Once the sensor is triggered, the forward motion is followed in sequence in the program by

statements to initiate backward motion, which is then reflected in the model. In contrast, the control flow of the version in Figure 2 seems to have no direct link with the model's behavior; the program appears to do nothing further once the motors have been started, yet the model does behave as required.

```
/**
 * A model with left and right motors,
 * plus a single touch sensor.
 */
public class OneBumper {
   private Motor left = Motor.A,
                 right = Motor.C;
   private Sensor bumper = Sensor.S1;

   /**
    * Run forward until an obstacle is
    * hit. Then reverse.
    */
   public void run(){
      bumper.setTypeAndMode(
        SensorConstants.SENSOR_TYPE_TOUCH,
        SensorConstants.SENSOR_MODE_BOOL);
      bumper.addSensorListener(
        new SensorListener(){
           public void stateChanged(
                   Sensor s,int oldValue,
                   int newValue){
              if(newValue == 1){
                 // We hit a wall.
                 left.backward();
                 right.backward();
              }
           }
      });

      left.setPower(3);
      right.setPower(3);
      left.forward();
      right.forward();
   }
}
```

Figure 2: Attaching an event listener to a sensor

What we have is an illustration that the model operates as a separate process that is truly concurrent with the execution of its driving program. This is one of the most significant differences between real models and the sort of simulated worlds discussed in Section 3; simulated models tend to take discrete steps in synchrony with the control flow and method calls of their driving programs. For students who are trying to build mental models of program execution, these differences between program flow and model behavior, or real worlds and simulated worlds, may present difficulties.

It is worth noting, however, that there are strong similarities between this event-driven style of programming physical models and the programming of graphical user interfaces based on Java's AWT and Swing frameworks [1]. With GUIs, a user interacting with buttons, menus, etc., functions as the independent process. This concurrency is managed in both scenarios by the presence of separate threads of execution monitoring for events, and notifying listeners when they occur.

6 Object-Oriented Issues

When teaching object-oriented analysis and design, we encourage students to identify the natural classes and objects that exist within a particular scenario. Of course, the 'correct' identification of these depends strongly upon the level of abstraction we wish them to consider, and how far we plan to take the design through to implementation. This can be hard for students to deal with. For instance, if we wish to model a vehicle such as a car, is it necessary to model wheels, axles, pistons, petrol tank, windows, and so on? A big advantage of using a physical model that we are really going to program is that there is a much stronger focus for the modeling process – the motors are probably much more relevant than the individual wheels, for instance.

Having identified the natural classes that arise from a particular model, we would probably like the students to then create objects of those classes in their own programs. It is here that some stylistic issues of concern might arise from the programming examples we have used so far.

- In neither example have we created any device-related objects. In both, references to pre-created motor and sensor objects are obtained via static references. This may not be the best pedagogic approach in an objects-early course.
- The touch sensor of the model has had to be mapped to a generic `Sensor` object, and it is necessary to tell that object what type it actually represents. Normal object-oriented style would require that an object's type should be an inherent part of its class definition.

In the next section, we seek to address some of these concerns.

7 Working with Alternative APIs

The comments towards the end of the previous section are not intended as a criticism of the leJOS platform API. We recognize that the main reasons behind its design are pragmatic; the amount of memory available within the RCX is only 32K bytes. If sufficient memory is to remain available for user programs, it is important that the firmware and general API be as lean as possible, yet still provide a practical general-purpose programming environment. Our requirements of the API happen to be constrained by the highly specialized desire to present a pedagogically clean object-oriented programming environment. An API that purely satisfies these requirements runs the risk of being too bloated to allow practical programs to be written with it. However, we believe that one of the best features of the leJOS project is that the platform is sufficiently well defined that it is viable

to develop custom APIs to suit individual teaching environments and the particular abstractions to be pursued in different courses.

Figure 3 presents a version of the running example using one such alternative API [2], designed to support an objects-early approach to Java.

```
/**
 * A model with left and right motors,
 * plus a single touch sensor.
 */
public class OneBumper {
   private Motor left = new Motor('A'),
                 right = new Motor('C');
   private TouchSensor bumper =
                 new TouchSensor(1);

   /**
    * Run forward until an obstacle is
    * hit. Then reverse.
    */
   public void run(){
      bumper.addSensorListener(
        new BooleanSensorListener(){
           public void stateChanged(
                   boolean oldValue,
                   boolean newValue){
              if(newValue){
                 // We hit a wall.
                 left.backward();
                 right.backward();
              }
           }
      });

      left.setPower(3);
      right.setPower(3);
      left.forward();
      right.forward();
   }
}
```

Figure 3: Programming with an alternative API

This API differs from the standard one in that instances of the `Motor` class may be created and a small set of typed sensor classes are also available. In addition, alternative listener interfaces permit sensor values to be notified to an application as either integer or Boolean values.

In order to assess the memory impact of the different approaches, Table 1 presents a comparison of the static size and remaining free space for each of the three versions of the programming example we have discussed. It is clear from these that the use of event handlers is more memory hungry than the use of polling. In addition, execution using the alternative API has consumed a further approximately 800 bytes of free space at runtime.

Version	Static size	Free space
Figure 1	3031	6046
Figure 2	3124	5444
Figure 3	3999	4632

Table 1: Static size and remaining free space in the different versions (values in bytes).

8 Further Issues

When developing course material based around the RCX, there are some further practical issues that are worth taking into account.

In traditional programming environments, when things are not working as expected we are accustomed to inserting extra print statements or using a debugger in order to track down the problem. With only a single small LCD line on the RCX to print to, lack of feedback can be a problem, unless you are prepared to communicate back to the host PC via the IR device! While as a teacher one might hope that this would make students more systematic in their approach to problem solving [17], students may simply find it frustrating.

Allied to the debugging issue is the length of time taken by the edit-compile-upload-run cycle. With upload speeds of the order of 100 bytes per second, the cycle can be significantly longer than with purely PC-hosted examples. It is amusing to reflect that the availability of a simulator would help here! It might also be worth noting that the time taken to code and debug examples in the Karel simulator was one reason why Pattis felt that students might better spend their time reading the book and working through the problems on paper [14].

In Section 5, we noted some of the perceptual differences between physical models and their simulated counterparts. Further differences arise from the inexact nature of a physical model's movements and the environment in which it operates. For instance, two motors driving independent wheels are unlikely to match each other exactly in their output, causing a model to drift unpredictably from a straight line. This is impossible to allow for in software without mechanical assistance from a combination of differential gearing and rotation sensors [8]. Furthermore, typical actions that are relatively easy to program in simulation software – such as turning a Turtle through a fixed angle – can be hard to configure in a physical model.

9 Conclusions

It has long been recognized that computer-controlled models are a useful aid in teaching introductory programming. Physical models provide an interesting alternative to simulated models which are commonplace. LEGO MINDSTORMS models are a particularly convenient way to build physical programmable models without having to know anything about hardware. The

availability of replacement firmware, such as leJOS, means that models built around the RCX processor can be programmed using Java – an increasingly common introductory programming language. Aside from the programming aspects, this environment certainly provides nice illustrations of two fundamental concepts that are often associated with Java; bytecode portability and the programmability of small devices.

In this paper, we have considered some of the pedagogic issues that arise from using the RCX in combination with Java for introductory programming. We have seen that there are some significant differences between simulated models and their physical counterparts. The concurrent and event-driven nature of physical models best suits a programming style that may be considered relatively advanced for introductory students, although there are similarities with the style used to program graphical user interfaces, which is regularly undertaken on introductory courses.

We also believe that there is a risk that good object-oriented programming style could be distorted by the pragmatics and physical limitations of the RCX environment. However, we have suggested that there is good scope for developing custom APIs that better suit the pedagogic requirements of particular courses, in order to mitigate these disadvantages.

On balance, we believe that is better to use these models to enhance and support an introductory programming course rather than as the basis for a whole course.

10 Acknowledgements

I am particularly grateful to Michael Caspersen (University of Aarhus) for demonstrating a Java-programmed RCX turtle, which sparked my initial interest in these models. In preparing this paper, I valued the discussions I had with a number of people, including Roger Glassey (University of California, Berkeley), Janet Linington (University of Kent at Canterbury), and Bill Margolis (National University of Samoa), all of who are using the RCX with Java in their teaching.

References

[1] Barnes, David J. *Object-Oriented Programming with Java: An Introduction*. Prentice-Hall, 2000.

[2] Barnes, David J. An API for the Lejos platform, intended for introductory Java instruction. Online. Internet. [August 22, 2001]. Available WWW: http://www.cs.ukc.ac.uk/people/staff/djb/rcx/

[3] Baum, Dave. *Definitive Guide to LEGO MINDSTORMS*. Apress, 2000.

[4] Becker, Byron Weber. Teaching CS1 with Karel the Robot in Java, in *Proceedings of the 32nd SIGCSE* (Feb 2001), ACM Press, 50-54.

[5] Erwin, Benjamin, *Creative Projects with LEGO Mindstorms*, Addison-Wesley, 2001.

[6] Hempel, Ralph. pbFORTH. Online. Internet. [August 22, 2001]. Available WWW: http://www.hempeldesigngroup.com/lego/pbFORTH/

[7] Knudsen, Jonathan B. *The Unofficial Guide to LEGO MINDSTORMS Robots*, O'Reilly, 1999.

[8] Knudsen, Jonathan B. The Straight and Narrow, O'Reilly Network, 2000. Online. Internet. [August 22, 2001]. Available WWW: http://www.oreillynet.com/lpt/a/network/2000/05/22/LegoMindstorms.html

[9] LEGO. LEGO MINDSTORMS Official site. Online. Internet. [August 22, 2001]. Available WWW: http://mindstorms.lego.com/

[10] LUGNET. LEGO Users Group Network. Online. Internet. [August 22, 2001]. Available WWW: http://www.lugnet.com/

[11] Nelson, Russell, LEGO MINDSTORMS Internals. Online. Internet. [August 22, 2001]. Available WWW: http://www.crynwr.com/lego-robotics/

[12] Noga, Markus L. *legOS*. Online. Internet. [August 22, 2001]. Available WWW: http://www.noga.de/legOS/

[13] Papert, Seymour. *MINDSTORMS: Children, Computers, and Powerful Ideas*, The Harvester Press Ltd, 1980.

[14] Pattis, Richard E. *Karel the Robot: A Gentle Introduction to the Art of Programming*, John Wiley & Sons, 1981.

[15] Solorzano, Jose. leJOS: Java based OS for Lego RCX. Online. Internet. [August 22, 2001]. Available WWW: http://lejos.sourceforge.net/

[16] VanderBijl, Ryan. Lego Mindstorms Robotics. Online. Internet. [March 5, 2002]. Available WWW: http://cs.calvin.edu/CS/research/robots/Ryan/

[17] Wolz, Ursula. Teaching Design and Project Management with Lego RCX Robots in *Proceedings of the 32nd SIGCSE* (Feb 2001), ACM Press, 95-99.