AC 2009-573: CEMENTING ABSTRACTION WITH A CONCRETE APPLICATION: A FOCUSED USE OF ROBOTICS IN CS1

Alexander Mentis, United States Military Academy

Charles Reynolds, United States Military Academy

Donald Abbott-McCune, United States Military Academy

Benjamin Ring, United States Military Academy

Cementing Abstraction with a Concrete Application: A Focused Use of Robots in CS1

Abstract

Teaching abstraction, modularity, and encapsulation, as well as the essential skill of reading, understanding, and using code generated by other programmers is important in an introductory programming course (hereafter referred to as CS1). A problem often encountered, however, is that the details of the underlying code hidden by good and interesting abstractions are often too complex to be used to teach beginning programmers to read and understand code written by others. This paper introduces an easy-to-use Ada wrapper package for the iRobot® Create platform's Open Interface specification that is useful for illustrating the topics above, but it is implemented with code that is also relatively easy for beginning programmers to read. The accessibility of the underlying code further allows for its easy use in exploring more advanced concepts in follow-on CS and computer engineering courses. We believe our package and our approach toward its use support the successes other educators have had in integrating robotics into their curricula and overcome some of the difficulties that have been encountered.

1. Introduction

The motivation to create a wrapper for the iRobot® Create's serial command Open Interface specification initially came from a desire to improve the way we teach abstraction, modularity, and encapsulation in our CS1 curriculum. In addition to the treatment we give to those topics during normal lecture hours, our students also participate in a two-hour lab, with a graded take-home portion, in which they are expected to use an instructor-provided package to solve a problem. Our CS1 course assumes no prior programming knowledge other than a broadly-scoped information technology course taken by all our freshmen. The CS1 course is taught using Ada, so that is the language we used for our wrapper package. Ada was convenient to use for that reason, but we also believe the use of Ada lowers some of the barriers to using robots in the curriculum that others have encountered, and we discuss this later. Because of the way Ada provides encapsulation, Ada packages consist of two files: a specification, which describes the package's API; and a body, which contains the package's implementation.

Previously, the lab on packages typically consisted of a block of instruction focusing on the contents of the specification and how to use it, followed by a brief exposure of the "scary" implementation the API was altruistically hiding. The capabilities the package usually provided, while useful, were admittedly less than compelling, and the implementation was rarely, if ever, looked at again. The benefits of introducing our Open Interface wrapper package into this lab are three-fold: it serves to present a context for the discussion of abstraction and encapsulation that is more interesting to our students, implementing algorithms to control robots reinforces numerous basic programming and engineering skills appropriate to cover in CS1, and, most notably, the hidden details of the implementation are still simple enough to be used later in the CS1 course, or elsewhere in the curriculum, to illustrate more advanced programming and hardware concepts. It is this last point that we feel has been largely overlooked in other robotics integrations.

2. Background

The use of robots in CS1 to pique students' interests in computer science and to exemplify basic algorithmic and programming concepts has been well explored. During the five-year period 2000-2004, there was an explosion of interest in using robotics in CS1. Much of the argument for this was based on responding to the declining enrollment in computer science by making CS1 more engaging. It was thought that CS1 would be more exciting if it addressed real world problems that beginning students felt were meaningful and relevant, and it was argued that robotics could provide this.[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] Chilton[12] is a good, representative, and recent presentation of this argument and describes the use of a robot assignment as a single lab exercise in CS1. Other examples of robot integration include Balch[1], Blank[13, 14], Dodds[2], Fagin[3, 4, 15], Challinger[16], Danyluk[11], and Ladd[17].

Much of the early work used the LEGO® Mindstorms® robot and includes Patterson-McNeill[18], Lawhead[19], and Schep[20]. More recently, numerous descriptions of CS1 courses that use LEGO® Mindstorms® have appeared, including Hasker[5], Imberman[6], van Delden[8], and Wolz[10]. McNally describes the use of LEGO® Mindstorms® in a CS2 course[21], and Stevenson discusses integrating LEGO® Mindstorms® with a web cam to introduce image processing into undergraduate computer science[22].

Since 2004, our computer science program has attempted to achieve many of the same pedagogical goals as the above robot-based curricula through the use of an Ada TurtleGraphics package, which gives students and entry-level way to create programs that include graphics. Namely, the objectives were to maintain student motivation in the program and teach fundamental computer science concepts to beginning programmers.[23] Since educational robots have become less expensive, however, we have begun to explore ways to update our CS1 curriculum to recognize the increased influence of robotics in the field of computer science and to increase the interdisciplinary ties between our computer science, electrical engineering, and information technology programs.

In many ways, the concepts we have been teaching with TurtleGraphics overlap with the concepts others have taught with robots. Both paradigms easily illustrate fundamental CS1 topics such as basic programming[1, 9, 23, 24], algorithms (sequence, selection, and iteration)[3, 6, 19, 20, 23], modularity[15, 6, 20, 23], and abstraction[23, 9, 17]. In addition to these topics, though, the inclusion of robotics adds context for the discussion of event-driven computing[15, 19]; practical issues such as wireless communication latency, sensor sensitivity, and wheel slippage[15, 19, 7, 10]; and real-time and concurrency issues[19, 7]. While in-depth exploration of most of these additional topics must wait until the students are further along in their studies, a gentle introduction through the use of the robot and exploration of the software interface in an introductory course may make the learning curve in later courses less steep and set the stage for more efficient learning. Unfortunately, most of the previously mentioned solutions that create a simple-to-use robot control interface for the beginning programmer also appear to have highly complex implementations, making them unsuitable for the beginning programmer to read, explore, and learn from.

3. Design

We designed our wrapper package with an eye toward teaching students how to understand and re-use modular code created by others and also toward providing the potential for deeper exploration of the package details.

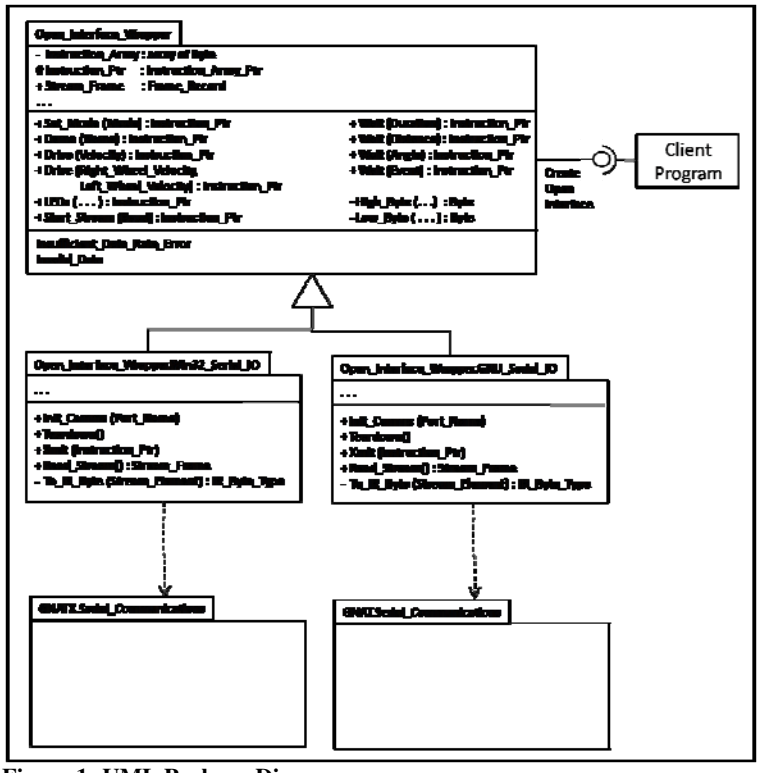

Figure 1: UML Package Diagram

Our primary inspiration for this package was to demonstrate the benefits of modularity and abstraction. While our program does not teach object-oriented programming in CS1, the design of our Open Interface Wrapper package is shown here in Figure 1 as a UML package diagram for ease of illustration.

3.1 Modularity

The package demonstrates modularity by encapsulating all of the subprograms required to communicate with the Create robot. The required package subprograms themselves can be placed into two categories: those that simply generate Open Interface opcodes and their associated data, and those that transmit or receive serial data to or from the Create. The first category of subprograms is specific only to the format of the Create's Open Interface, which describes the byte value of the opcode and the format of the data bytes it requires. The second category, however, is operating system-specific.

We take advantage of the fact that the first category is common to all operating systems by putting them in the parent Open Interface wrapper package, and we place the subprograms belonging to the second category into operating system-specific child packages of the parent package. When a client program links the child package for the target operating system, the child package also inherits all of the functionality of the parent package. As we will describe later, this design is especially apt for driving home to CS1 students the power of well-designed modularity.

At this point it should be noted that, as indicated in Figure 1, our package supports both the Windows and GNU/Linux operating systems through a dependency on the GNAT Ada compiler's GNATX.Serial_Communications and GNAT.Serial_Communications packages, respectively. These packages are available with the GNAT GPL 2008 distribution.

3.2 Abstraction

Our package provides public Ada functions for most of the Create Open Interface opcodes, and we have plans to implement more functionality in future versions. Each function accepts a set of parameters sufficient to generate the data bytes required by its opcode and returns a pointer to a dynamically-sized array of bytes containing the opcode and data bytes. This pointer can then be sent as a parameter to the Xmit procedure, provided in the linked child package, which streams the bits to the Create in a manner compatible with the target operating system.

By the point in the course that our students receive this lab, they have seen fixed-length arrays, but they have not seen dynamically-sized arrays or pointers. The resultant array that contains the instruction bytes is made private so the students cannot individually access the instruction array or its elements – they must create the array through the provided functions and pass it to the Xmit procedure via a pointer. Even though they do not have knowledge of pointers, the students are still able to use pattern matching to understand that the package functions return values of type Instruction_Ptr, and that is exactly the type of the parameter required by Xmit. Ada's strong typing and extensive type/subtype definition capabilities serves to further prevent the use of improper parameter values. As shown in Figure 1, the Instruction_Ptr is "protected" in the sense that the students are allowed to use the Instruction_Ptr type provided by the package, but they cannot access any part of its implementation.

In addition to hiding the details of pointers from the users of the package, the following technical aspects of creating an instruction are hidden as well: the conversion of a 16-bit integer into high and low bytes, and the manner in which the inverse of an Event is calculated for the Wait function. The high and low bytes of a 16-bit integer are created for packaging in the bit stream by two private functions, High_Byte and Low_Byte. These helper functions can be called by any instruction-creating function that needs them, leaving the user of the package free to send a single integer value to the function and letting the function break it up if necessary. Similarly, modular arithmetic is used to calculate the data byte values when the Wait(Event) function is instructed to wait for the inverse of an event, so the user does not need to calculate or look up the inverted event value. The user simply specifies the desired event and sets a flag if he or she requires its inverse.

The details of reading a bit stream from the robot are hidden from our students, too. Details of the Read function that are too advanced for the point in the course at which we use this lab are: achieving frame lock on a continuous bit stream, converting a byte value into an enumerated type value, converting two unsigned bytes into a signed 16-bit integer, creating a record to store all the received values, and dealing with concurrent read/write access to a shared record structure. As was the case in the use of pointers, the use of records several lessons earlier than their formal introduction in the course is not a detractor. Instead, it provides foreshadowing and motivates their later study in depth. Since the record type is provided in the package, students simply need to declare a variable of the provided type and be shown the Record_Name.Field_Name notation for accessing the data.

This section has focused on the usefulness of our package for hiding the implementation details. In the following sections, we show that our implementation is also beneficial for teaching concepts that are covered later in the course.

4. Implementation

Aside from implementing modularity and abstraction, there were a number of other basic programming concepts and features of the Ada language that we wanted to ensure we included in the Open Interface wrapper package to support our curriculum. Like the currently used TurtleGraphics package, which we are keeping in our curriculum and augmenting with robotics, our Open Interface wrapper has subprograms with default parameter values, user-defined exceptions, and overloaded subprogram names.[23] Unlike TurtleGraphics, however, the abstracted code in our wrapper package is much more accessible to beginning programmers.

It was well-recognized at the initial introduction of the TurtleGraphics package that the GtkAda GUI API it is based on was "too large and idiomatic for beginning programmers to comprehend."[23] Indeed, the "extensive use of a type-safe object hierarchy and callbacks . . . [that was] . . . beyond the scope of CS1"[23] was among the primary motivators for the creation of the TurtleGraphics package in the first place – to hide that complexity. In contrast, our Open Interface wrapper package implementation primarily uses unconstrained (dynamically-sized) arrays, access types (pointers), modular (unsigned) types, and records – all topics that are appropriate for CS1. In addition, our package uses some low-level data representation handling that, while not necessarily appropriate for CS1, could be illustrative to computer engineers. Thus, we have been able to keep all the best pedagogical features of TurtleGraphics while using mostly CS1-level code.

4.1 Default Parameter Values

Two functions in our package have default parameter values: LEDs and Wait(Event). The more interesting of the two is the Wait function. This function provides the Open Interface opcode to either wait for the occurrence of an event (such as a left bumper activation) or the absence of the event (its inverse). An enumerated type, listing the different events that can be detected, is defined by the package. The assumption is that it will be more common to wait for an event rather than its absence, so while the function can take two parameters (the Event and whether or not the user wants the inverse), the inverse parameter is assigned the value of False by default and only required in the event that the inverse event is desired. One of the benefits of using this approach, as shown in Figure 2 is that it leads to a relatively simple calculation of the byte value for the opcode data based on the position of the event in the enumeration and whether or not the user wants the inverse. Thoughtful instruction design on the part of the computer engineer makes this possible. Technically, the complementary values are just the unsigned

```
if Inverse then
   Instruction (2) := Interfaces.Unsigned_8 ((-(Event_Type'Pos (Event) + 1)) mod 256);
else
   Instruction (2) := Interfaces.Unsigned_8 (Event_Type'Pos (Event) + 1);
end if;
```

Figure 2: Event Value Calculation

inverses of each other. Pedagogically, this code can be used to simply demonstrate "wrap-around" arithmetic to CS1 students and show them that the calculation does, in fact, change the values to their inverse as specified in the Open Interface specification.

4.2 User-Defined Exceptions

Our package defines two user-defined exceptions: Insufficient_Data_Rate_Error, raised when the baud rate of the serial connection is too low to support receiving the amount of data contained in the sensor packets within 15ms (the frequency at which the Create's updates sensor data when streaming), and Invalid_Data. Our package handles the first exception when necessary; however, it can be educational to point out to the students the reason why it is necessary and use it to start a discussion about the considerations one must make when writing real-time applications. The second exception is made available for students to use in handling cases where the Read_Stream function returns an invalid frame. Testing for the condition and raising the exception is left to the students.

4.3 Overloading

Figure 1 shows that the Drive and Wait functions are overloaded. There are two interesting points to be made with students about these particular overloaded functions. The first is simply to demonstrate that it is possible to give the same name to two different subprograms with similar behavior. It should be easy for them to see how Ada distinguishes between the two Drive functions, since they have different numbers of parameters. It may not be so obvious how Wait(Angle), Wait(Distance) and Wait(Duration) can be differentiated, though, since they all take one parameter, the range of which is a subset of type Integer. This can be a good way to have a more in-depth discussion about method signatures and what the compiler needs in order to differentiate a signature. In this case, since the Integer subsets of the three parameters are not disjoint, a simple Ada subtype is not sufficient to differentiate them – one has to define a "new" Integer type of the appropriate range for each of the three parameters. Figure 3 shows what that looks like in our package.

```
type Duration_Type    is new Integer range 0 .. 255;       -- 10ths of a second
                                                           -- (15ms resolution)

type Distance_Type    is new Integer range -32_767..32_768; -- mm
                                                              -- positive values
                                                              --  are forward
                                                              --  distances
                                                              -- negative values
                                                              --  are reverse
                                                              --  distances

type Angle_Type       is new Integer range -32_767..32_768; -- mm
                                                           -- positive values are
                                                           --  counterclockwise
                                                           -- negative values are
                                                           --  clockwise
```

Figure 3: Definition of New Integers

4.4 Unconstrained Arrays

An Open Interface instruction consists of a one-byte opcode, followed by zero or more data bytes. Some instructions, such as Song, which takes in a series of notes and their durations, are arbitrarily long. While our package does not currently support song creation, we wanted to make future expansion of the package as easy as possible. Since it is impossible to know a priori how many elements an Instruction_Array will contain, we used dynamically-sized arrays (called unconstrained arrays in Ada) to implement them. Each functional implementation of an Open Interface instruction creates an appropriate Instruction_Array from the opcode listed in the Open Interface specification and the parameters passed to it. We cover unconstrained arrays in our CS1 course approximately three quarters of the way through the semester. Figure 4 is an example of code that can be used to show how the overloaded direct Drive function creates a dynamically-sized instruction based on user-desired wheel velocities for each wheel. Note the use of the High_Byte and Low_Byte helper functions to break the velocity parameter values into component high and low bytes.

```
function Drive (Right_Wheel_Velocity,
                Left_Wheel_Velocity  : Velocity_Type)
               return Instruction_Ptr is

   Instruction : Instruction_Ptr;

begin -- Drive (Direct)

   Instruction := new Instruction_Array(1..5);

   Instruction(1) := 145;
   Instruction(2) := High_Byte(Integer(Right_Wheel_Velocity));
   Instruction(3) := Low_Byte (Integer(Right_Wheel_Velocity));
   Instruction(4) := High_Byte(Integer(Left_Wheel_Velocity));
   instruction(5) := Low_Byte (Integer(Left_Wheel_Velocity));

   return Instruction;

end Drive;
```

Figure 4: Drive (Direct) Implementation

4.5 Access Types

Figure 4 also shows a typical function's mechanism for returning the Instruction_Ptr required by the Xmit procedure. Using a pointer to send an instruction to Xmit allows Xmit to remain ignorant of the size of the instruction it is receiving.

4.6 Modular Types

Unsigned integers are implemented as "modular types" in Ada. Using modular types enables bitwise manipulation of the values and various bit shifting operations, provided by Ada's Interfaces package. Figure 5 shows the implementation of the High_Byte helper function, which

```
function High_Byte(Value : Integer) return Interfaces.Unsigned_8 is

   Word : Interfaces.Unsigned_16 := 0;

begin -- High_Byte

   Word := Interfaces.Unsigned_16((Value) mod 2**16);
                          -- convert to unsigned value (congruent mod 16)
                          -- mask select upper eight bits
                          -- shift out the lov byte
   return Interfaces.Unsigned_8(
     Interfaces.Shift_Right(
       Interfaces.Unsigned_64(Word and 2#11111111_00000000#), 8
     )
   );

end High_Byte;
```

Figure 5: High_Byte Implementation

demonstrates both of these operations. The coverage of modular types is an optional lesson in our CS1 curriculum, taught at the instructor's discretion if the class appears to have a firm grasp of the fundamentals. This code can be used to provide perspective on their application.

4.7 Records

As we noted in section 3.2, a frame of data read from the serial stream is stored as a record containing a field for each sensor value we are expecting. There is nothing especially notable about the way we implement the record. It merely gives us a way to demonstrate one of many practical uses for a data structure our students learn about during the course.

4.8 Low-Level Representation of Data

The Create Open Interface assigns byte values to the Create infrared remote control and home base signals it receives as shown in Figure 6. As one can see, they are neither contiguous, nor do they start at zero. To convert the received byte data to an enumerated type would normally require a test and assign selection construct. In Figure 6, however, we use an advanced feature of Ada that allows us to associate each enumerated value with the appropriate byte representation. When a byte is received, the value is converted directly using Unchecked_Conversion.

```
for IR_Byte_Type use (-- sent by iRobot remote control
                      Left                        => 129,
                      Forward                     => 130,
                      Right                       => 131,
                      Spot                        => 132,
                      Max                         => 133,
                      Small                       => 134,
                      Medium                      => 135,
                      Large_Clean                 => 136,
                      Pause                       => 137,
                      Power                       => 138,
                      Arc_Fwd_Left                => 139,
                      Arc_Fwd_Right               => 140,
                      Drive_Stop                  => 141,
                      -- sent by iRobot scheduling remote
                      Send_All                    => 142,
                      Seek_Dock                   => 143,
                      -- sent by iRobot home base
                      Reserved                    => 240,
                      Force_Field                 => 242,
                      Green_Buoy                  => 244,
                      Green_Buoy_And_Force_Field  => 246,
                      Red_Buoy                    => 248,
                      Red_Buoy_And_Force_Field    => 250,
                      Red_And_Green_Buoy          => 252,
                      All_Buoys_And_Force_Field   => 254,
                      No_IR_Signal                => 255);
```

Figure 6: Low-Level Data Definition

5. Analysis

Donald Knuth, speaking about learning to program, said, "The code, once I saw how it happened, was inspiring to me. Also, the discipline of reading other people's programs was something good to learn early. ...[U]nderstand[ing] their thought processes ... [was] the best way ... to get ... past the stumbling blocks."[25] We agree that there is tremendous benefit to exposing students to others' code, both to generate excitement about programming and to see good examples. For these tasks, the code should be neither too trivial, nor overwhelming for the student at his or her level of experience. We believe the Open Interface wrapper package described above fits this description for students in CS1.

5.1 Teaching Abstraction

Our package serves the same function as other packages we have used in our abstraction-focused lab. At the most basic level, the essential skill of learning how to use modular code created by others comes down to being able to read and understand an API. As we described earlier, Ada's mechanism for describing a package's API is contained in the package's specification file. This file lists the data types and exceptions defined by the package, the subprograms it provides, the parameters of those subprograms (including any default values), and the return types of the functions. Our Open Interface wrapper package provides a rich API for students to read and learn. It is not overwhelming to them, though, because most of the functions behave similarly: they create an instruction for a particular robot behavior. Therefore, at the point in the course when we use this package to teach abstraction, we continue to cover the API in detail and the implementation only briefly.

5.2 Teaching Modularity

The use of relatively simple child packages in our package makes a nice contribution to the modularity portion of the lab that we haven't included before. We use the Red Hat Enterprise Linux 5 operating system in our classroom lab, but the students use the Windows XP Professional operating system when they take the robots back to their rooms to work on the graded assignment. Even though the two operating systems handle serial port naming and communication differently and depend on completely different serial communication packages, we can show the students how easy it is to add child packages to handle the differences of a new operating system without changing the package interface. The same client programs are used on both operating systems with no change other than linking the correct child package to the client.

```
-------------------------------------------------
-- Gtk globals for communication among callbacks --
-------------------------------------------------
User_Task : Task_Id := Null_Task_ID; -- Task that sets up world.
Pixmap : Gdk_Pixmap;           -- Pixmap used as backing store.
Window : Gtk_Window;
Pausing : Boolean := False;
Vbox : Gtk_Box;
Drawing_Area : Gtk_Drawing_Area;
Idle_Function_Id : Idle_Handler_Id;
Pause_Timeout_Function_Id : Timeout_Handler_Id;
Draw_Timeout_Function_Id : Timeout_Handler_Id;
Width, Height : Gint;
Gc : Gdk_Gc;
type Gdk_Color_Array_Type is array(Color_Type) of Gdk_Color;
Colors : Gdk_Color_Array_Type;
type Rgb_Type is
   record
      R, G, B : Guint16;
   end record;
type Rgb_Array_Type is array(Color_Type) of Rgb_Type;
Max_Rgb_Val : Guint16 := Guint16(2.0 ** 16 - 1.0);
Med_Rgb_Val : Guint16 := Guint16(Float(Max_Rgb_Val) * 7.0 / 8.0);
Dk_Gray_Val : Guint16 := Guint16(Float(Max_Rgb_Val) * 1.0 / 4.0);
Gray_Val    : Guint16 := Guint16(Float(Max_Rgb_Val) * 2.0 / 4.0);
Lt_Gray_Val : Guint16 := Guint16(Float(Max_Rgb_Val) * 3.0 / 4.0);
Rgb_Vals : Rgb_Array_Type :=
  (Black      => (         0,          0,          0),
   Red        => (Med_Rgb_Val,          0,          0),
   Green      => (         0, Med_Rgb_Val,          0),
   Yellow     => (Med_Rgb_Val, Med_Rgb_Val,          0),
   Blue       => (         0,          0, Med_Rgb_Val),
   Magenta    => (Med_Rgb_Val,          0, Med_Rgb_Val),
   Cyan       => (         0, Med_Rgb_Val, Med_Rgb_Val),
   Dark_Gray  => (Dk_Gray_Val, Dk_Gray_Val, Dk_Gray_Val),
   Gray       => (   Gray_Val,    Gray_Val,    Gray_Val),
   Light_Gray => (Lt_Gray_Val, Lt_Gray_Val, Lt_Gray_Val),
   White      => (Max_Rgb_Val, Max_Rgb_Val, Max_Rgb_Val));
```

Figure 7: GtkAda Callback Communication

5.3 Understanding the Implementation

As our implementation section shows, we have kept the internals of our package simple enough that we can encourage our students to look at the underlying code. Our implementation of High_Byte, shown in Figure 5, is about as complex as it gets, requiring the use of four topics we don't

normally cover in our CS1 course: unsigned integers defined in the Ada Interfaces package, modular arithmetic, bitwise masking, and bit shifting. While we choose not to cover those topics due to time constraints in the semester, one can see that they are not so complicated that the interested CS1 student with a little experience with those topics in other languages or a short visit during office hours couldn't easily grasp.

Contrast this with the internals of the TurtleGraphics package we also use in our course. Figure 7 shows the setup of global variables required for communication among GtkAda callbacks.[23] The reader of this package code must become familiar with twenty-three Glib, Gdk, and Gtk libraries. It also makes extensive use of multithreading and queues. While the number of lines in each package (~800 for TurtleGraphics versus ~750 for Open Interface wrapper) is similar, this is deceptive, since our Open Interface package includes extensive comment blocks. The number of executable statements in the Open Interface wrapper package is actually much lower than the number of statements in TurtleGraphics. We love giving our students an easy way to make graphic-based programs, but we include a sample of the TurtleGraphics code here as a prime example of why most interesting implementations of abstraction are too complex internally to provide CS1-level examples.

The fact that our Open Interface wrapper abstraction is mostly hiding topics that we will eventually cover in detail, on the other hand, is of great pedagogical benefit. Using the package to control the robots early in the course provides a cognitive hook for those topics when we get to them later in the course. When we introduce a new topic, we often have a class discussion about how it might be used in a practical way. This package provides many practical uses for these more advanced concepts. Through the use of the package in the lab, students will already have an appreciation of some of these applications and the considerations they require.

6. Evaluation

We have compared our robotics implementation and its integration into our curriculum to what others have done. We see many areas where we have achieved success similar to those of others and some areas where we have overcome problems in robot integration noted by others. Our approach is scoped such that we do not require a robot for every student, we achieve several benefits from off-platform processing, we avoid special programming language subsets and complex cross-compilation "magic", we maintain code readability and accessibility for novice programmers, and we leave open the potential use of the platform for more advanced courses.

Some educators that have met with limited success incorporating robotics in their curriculum cite an inability to provide a robot to every student as a significant obstacle.[4, 7] If the robots are to serve as the primary programming platform for the entire course, we agree with Balch that it is ". . . essential . . . that every student should have his or her own robot".[1] In our CS1 course, though, we only use robots in one lab focused on modularity and abstraction. Like Hasker, we do not exclusively use robots to teach every topic in CS1.[5] Certainly, programming robots can also illustrate basic programming concepts, but we think that our focused application with the occasional exposure of an advanced topic's practical use is an appropriate and highly effective compromise between the need to create excitement for the discipline while recognizing that computer science is about more than just robotics.

By taking this approach, we mitigate the need for each student to have a personal robot, since the robot is more loosely integrated with the learning objectives, and the time span of the robot integration is much shorter. We assign one robot to each group of 2 – 3 students, and at our institution, the students are allowed to take the robots home for the duration of the assignment. If this is not a viable option for other institutions, there may be other options, such as staggering the labs so that different sections of the course need the robots at different times, or redefining the project so that the client can be developed with little direct interaction with the robot. Challinger[16] describes an example of the latter for which our package could easily be used.

In fact, since our client code using the package is not downloaded to the robot, but rather communicates wirelessly with the robot via Bluetooth, we maintain greater flexibility and potential reuse of this package across the curriculum, including use in CS2 courses, because our package can be used for both scripted and real-time applications. Some applications of robotics in the literature that our package could support (at least in part) include Chilton [12], Dodds [2], Imberman [6], McNally [21], Shep [20], van Delden [8], Wolz [10], and Challinger [16]. This is quite contrary to Weiss' evaluation of the Create, which he found to be rather inflexible and non-reusable. [9] Furthermore, the off-platform processing results in a shorter design-test-modify loop, noted in McNally [7] as a detractor in the use of LEGO® Mindstorms®. Finally, by processing off-platform, we see the same benefits as Balch in harnessing the processing power of the computer running the client program, easier debugging in an integrated development environment, and cross-platform compatibility with Windows and Linux through the programming language's support of serial communication.[1]

The benefits of using Ada and streaming serial communication straight to the robot are not limited to the above points. It also avoids many of the complex and time-intensive cross-compilation schemes and language subsets others have had to use. Fagin's use of Ada with LEGO® Mindstorms® required cross-compilation from a limited Ada subset to Not-Quite-C (NQC) that both constrained what the student could do in Ada and hid the low-level interaction with the robot.[3, 4, 15] Hasker encountered a number of disappointments with NQC, Bricx Command Center (BricxCC), and legOS for LEGO® Mindstorms®, including limited language capabilities, cryptic compiler errors, and complex environment dependencies. [5] This led him to develop a programming environment called HiC, a subset of C++. It's interesting to note that a good number of the features Hasker chose to leave out due to their complexity (bit manipulation, accessing arrays via pointers, and exceptions) [5] are exactly the same features we were so excited to be able to examine in our package when using Ada.

A significant drawback to both of the above cases, in our opinion, is that the abstractions turn the robot into too much of a black box. How the program actually communicates with the robot at a low level is therefore hard to explore. Since students do not actually build the Create, there are a limited number of opportunities to cover computer engineering topics with our package. Nevertheless, by making the details of the bitwise handling of opcodes part of our package, we provide a tangible example for some of the considerations computer engineers must make, such as designing instruction sets, assigning meanings to byte values, interpreting low-level data representations in code, determining frame lock on a continuous stream, and CRC checking to name a few.

In addition to these topics, there are also the practical difficulties of dealing with an analog world in a discrete manner. One lesson that our CS1 students quickly learn (and also noted by Dodds [2]) is that commanding a 90-degree angle does not guarantee a precise 90-degree turn in the real world. Another is the revelation of exactly how much data can be sent wirelessly to a robot between two key presses. Learning this on their own through observation and troubleshooting appears to ingrain the point in the students' minds more than simply telling them it is so during lecture.

7. Future Work

There are two areas in which our package requires more work. First, at the time of this writing, the package does not protect the shared sensor data from concurrent reads and writes. We would like to introduce multithreading so that a writer task can keep the sensor data current and the client program can use a reader task to get sensor values as needed. When this is complete, we intend to make the package publicly available. Second, while we have used this package for one semester in our CS1 program and appear to be getting good results, we would like to conduct formal assessment on the use of the package for the stated goals.

8. Conclusion

We have presented a wrapper package for the iRobot® Create's Open Interface specification. The design and implementation are intended to provide an excellent example of modularity and abstraction, but we believe it also provides the added benefit of being usable throughout a curriculum. Our relatively simple package implementation, compared to other solutions we have found in the literature, makes examination of the underlying code accessible to beginning programmers, and our foundation of the package on Ada using serial streams contributes to the flexibility of the package for many different applications, ranging from CS1 to more advanced studies.

Bibliography

1. Tucker Balch et al., "Designing Personal Robots for Education: Hardware, Software, and Curriculum," *IEEE Pervasive Computing*, vol. 7, no. 2, pp. 5-9, April 2008.
2. Zachary Dodds, "AI Assignments in a CS1 Course: Reflections and Evaluation," *Journal of Computing Sciences in Colleges*, vol. 23, no. 6, pp. 262-271, June 2008.
3. Barry Fagin, "Using Ada-based Robotics to Teach Computer Science," in *ACM SIGCSE Bulletin*, vol. 32, New York, 2000, pp. 148-151.
4. Barry Fagin and Laurence Merkle, "Measuring the Effectiveness of Robots in Teaching Computer Science," in *ACM SIGCSE Bulletin*, vol. 35, New York, 2003, pp. 307-311.
5. Robert W. Hasker, "An Introductory Programming Environment for LEGO MindStorms Robots," in *Midwest Instruction and Computing Symposium*, 2005.
6. Susan P. Imberman and Roberta Klibaner, "A Robotics Lab for CS1," *Journal of Computing Sciences in Colleges*, vol. 21, no. 2, pp. 131-137, December 2005.

7. Myles McNally, Michael Goldweber, Barry Fagin, and Frank Klassner, “Do Lego MindStorms Robots have a Future in CS Education?,” in *SIGCSE*, New York, 2006, pp. 61-62.
8. Sebastian van Delden and Wei Zhong, “Effective Integration of Autonomous Robots Into an Introductory Computer Science Course: A Case Study,” *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 10-19, April 2008.
9. Richard Weiss and Isaac Overcast, “Finding Your Bot-mate: Criteria for Evaluating Robot Kits for Use in Undergraduate Computer Science Education,” *Journal of Computing Sciences in Colleges*, vol. 24, no. 2, pp. 43-49, December 2008.
10. Ursula Wolz, “Teaching Design and Project Management with Lego RCX Robots,” in *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education* , New York, 2001, pp. 95-99.
11. Andrea Pohoreckyj Danyluk, “Using Robotics to Motivate Learning in an AI Course for Non-Majors,” in *AAAI 2004 Spring Symposium on Accessible Hands-on Artificial Intelligence and Robotics Education*, 2004.
12. John Chilton and Maria Gini, “Using the AIBOs in a CS1 Course,” in *American Association for Artificial Intelligence Spring Symposium - Robots and Robot Venues: Resources for AI Education*, 2007.
13. Douglas Blank, Lisa Meeden, and Deepak Kumar, “Python Robotics: An Environment for Exploring Robotics Beyond LEGOs,” in *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, New York, 2003, pp. 317-321.
14. Douglas Blank, Deepak Kumar, Lisa Meeden, and Holly Yanco, “The Pyro Toolkit for AI and Robotics,” *AI Magazine*, vol. 27, no. 1, pp. 39-50, Spring 2006.
15. Barry Fagin, “Ada/Mindstorms 3.0,” *Robotics & Automation Magazine, IEEE*, vol. 10, no. 2, pp. 19-24, June 2003.
16. Judith Challinger, “Efficient Use of Robots in the Undergraduate Curriculum,” in *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, New York, 2005, pp. 436-440.
17. Brian Ladd and Ed Harcourt, “Student Competitions and Bots in an Introductory Programming Course,” *Journal of Computing Sciences in Colleges*, pp. 274-284, 2005.
18. Holly Patterson-McNeill and Carol L. Binkerd, “Resources for Using Lego Mindstorms,” *Journal of Computing Sciences in Colleges*, pp. 48-55, 2001.
19. Pamela B. Lawhead et al., “A Road Map for Teaching Introductory Programming Using LEGO Mindstorms Robots,” in *ACM SIGCSE Bulletin*, vol. 35, New York, 2003, pp. 191-201.
20. Madeleine Schep and Nieves McNulty, “Use of Lego Mindstorm Kits in Introductory Programming Classes: A Tutorial,” *Journal of Computing Sciences in Colleges*, pp. 323-327, 2002.
21. Myles F. McNally, “Walking the Grid: Robotics in CS2,” in *ACM International Conference Proceeding Series; Vol. 165 Proceedings of the 8th Austalian conference on Computing education - Volume 52*, Darlinghurst, Australia, 2006, pp. 151-155.
22. Daniel E. Stevenson and James D. Schwarzmeier, “Building an autonomous Vehicle by Integrating Lego Mindstorms and a Web Cam,” in *ACM SIGCSE Bulletin*, New York, 2007, pp. 165-169.
23. Tanya Markow, Eugene Ressler, and Jean Blair, “Catch that Speeding Turtle: Latching onto Fun Graphics in CS1,” in *Proceedings of the 2006 Annual ACM SIGAda International Conference on Ada*, New York, 2006, pp. 29-34.
24. Barry S. Fagin, “Teaching Computer Science With Robotics Using Ada/Mindstorms 2.0,” in *Proceedings of the 2001 Annual ACM SIGAda International Conference on Ada*, New York, 2001, pp. 73-78.
25. Donald Knuth. Interview by Len Shustek. “The 'Art' of Being Donald Knuth,” in *Communications of the ACM*, vol. 7, no. 8, New York, 2008.