Why Structural Recursion Should Be Taught Before Arrays in CS 1*

Kim B. Bruce† , Andrea Danyluk, and Thomas Murtagh
Department of Computer Science
Williams College
Williamstown, MA 01267
{kim,andrea,tom}@cs.williams.edu

ABSTRACT

The approach to teaching recursion in introductory programming courses has changed little during the transition from procedural to object-oriented languages. It is still common to present recursion late in the course and to focus on traditional, procedural examples such as calculating factorials or solving the Towers of Hanoi puzzle. In this paper, we propose that the shift to object-oriented programming techniques calls for a significant shift in our approach to teaching recursion. First, we argue that in the context of object-oriented programming students should be introduced to examples of simple recursive structures such as linked lists and methods that process them, before being introduced to traditional procedural examples. Second, we believe that this material should be presented before students are introduced to structures such as arrays. In our experience, the early presentation of recursive structures provides the opportunity to reinforce the fundamentals of defining and using classes and better prepares students to appreciate the reasons to use classes to encapsulate access to other data structures when they are presented.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms

Algorithms, Design

Keywords

CS1, recursion

*Research partially supported by NSF CCLI grant DUE-0088895.
†currently on leave at UC Santa Cruz.

SIGCSE '05 Feb 23–27, 2005, St. Louis, Missouri, USA

1. INTRODUCTION

In the fall of 1999, we implemented a major update of the Williams College CS 1 course. The course is now taught using Java. With the support of the specially designed `objectdraw` library, this course takes an objects-first approach, uses truly object-oriented graphics, incorporates event-driven programming techniques from the beginning, and includes concurrency quite early in the course. As argued in our earlier papers [2, 1], the combination of object-oriented graphics, event-driven programming, and concurrency provides for a very interesting and pedagogically sound introduction to programming.

After the first few offerings of this course, we began to see a troubling pattern: in the second half of the course students were drifting away from thinking about object-oriented design issues. When we introduced arrays, students resisted encapsulating them in classes, instead simply making the arrays available globally as instance variables or passing them around as parameters.

For example, our first array lab involved presenting the Simon game, in which the computer generates longer and longer sequences of notes that the player must repeat. We emphasized to students that there should be a `Song` class in which the array of notes should be an instance variable, and which should support methods like `play`, `endOfSong`, etc. However, many students resisted creating this separate class to encapsulate the array. These students instead declared the array as an instance variable of the class that handled user interactions, and manipulated the array directly within the methods of that class.

Similar things happened with programs involving multi-dimensional arrays, strings, and files. We were concerned that we were teaching the first half of the course in an object-oriented style and the second half in a more procedural style.

Eventually, we devised a reordering of course topics that allows us to more firmly ingrain the object-oriented style of programming in ways that carry over into our introduction of arrays, etc. The key was to move the teaching of recursion from after arrays and strings to before these topics. Moreover, we changed from a focus on method-based recursion over the integers to a concentration on structural recursion. In the rest of this paper we discuss how we approach recursion and the associated benefits to having students design data structures in a more object-oriented style. While our own work took place in the context of using the `objectdraw` library, we believe that the same ideas can be used to improve any objects-early CS 1 course.

2. THE FIRST HALF OF THE COURSE

We begin our introductory course with the use of event-driven programming and truly object-oriented graphics. This use of graphics provides concrete examples of flexible and useful objects from predefined classes. The event-driven programming style allows students to create quite interesting programs using only very basic language constructs. The resulting methods tend to be relatively simple, while the implicit flow of control (e.g., from executing a method handling a mouse press to executing a method handling mouse drags to executing one handling a mouse release) allows students to focus on the manipulation of the objects without worrying about control structures.

After the introduction of conditional statements, students begin the design of multi-class programs. For example, students write programs that allows the user to create and drag around graphical objects generated by a class that they design. Objects on the screen can interact to produce interesting behavior. One such program we have assigned places pictures of two magnets in a window. When one magnet is dragged too close to the other, they either attract or repel, depending on which poles are close enough to interact.

We next introduce `while` loops. Threads (as extensions of our library's `ActiveObject` class) are introduced to provide a context in which simple `while` loops produce interesting behavior. In particular, `while` loops and threads can be used to produce animations as part of a program. As a first example, students write a program in which a click above a line results in the creation of a ball that then falls slowly down the screen. Points are scored if the ball falls entirely within a basket that is placed at a random position at the bottom of the window.

Later, our students write a Frogger game where mouse clicks are used to guide a frog across a four-line highway that includes cars moving in all of the lanes. In this example, there is a class for the frog, a class extending `ActiveObject` representing lanes of the highway, and another class extending `ActiveObject` representing the vehicles on the highway. From these examples, students get a great deal of experience building programs that involve several objects from different classes.

We then introduce Java `interface`s. We emphasize that if a variable's type is an interface, then the variable can hold objects generated by different classes that implement the interface. This flows into a relatively quick introduction to GUI interfaces using the Swing library.

At this point, we are 5 to 6 weeks into our 12 week semester. Topics to be covered in the second half of the term include one-dimensional and multi-dimensional arrays, strings, exceptions, files, recursion, and simple sorting and searching. Many of these topics are taught in ways fairly similar to those used in procedural languages, though of course the arrays generally hold objects from classes rather than just numbers.

Students learn best when fundamental ideas are reinforced by repetition and are used in different contexts. Because the object-oriented ideas no longer play as important a role in this portion of the course, student understanding of the rationale for object-oriented design seemed to slip, even in situations where the design of classes would provide substantial benefits. In the next section, we discuss how the introduction of recursion before arrays helped us keep the focus on the object-oriented approach.

3. TEACHING STRUCTURAL RECURSION BEFORE ARRAYS

Both arrays and recursive structures can be used to hold collections of objects. In typical introductory texts using procedural or object-oriented languages, arrays are introduced first, and only later are recursively defined structures like recursive lists introduced.[1] In fact many texts reserve their presentation of recursion for one of the final chapters, encouraging instructors to put this material off for discussion in a CS 2 course. We argue here that recursively defined structures are more object-oriented in flavor, and their relatively early introduction can help reinforce important ideas like dynamic method dispatch and the use of interfaces.

In addition, using structural recursion and presenting it before arrays makes it much easier to maintain student interest in the topic. Instructors who teach arrays before recursion have a difficult time convincing students of the value of using recursion. Most of the examples traditionally presented using recursion can easily be handled with loops or arrays or involve problems that appear to have little practical value to the students. On the other hand, at the point we now introduce recursion, students have not yet seen arrays or any other Java constructs that will allow them to represent collections of objects. Learning about recursive structures greatly expands the range of programming problems our students can solve by allowing them to write programs that manipulate such collections.

In this section we present a very simple example to illustrate our ideas. While we will use the `objectdraw` library to create a recursive graphic image, the underlying data structure is simply a specialized recursive list. Instructors can select other examples that fit in the context of their courses, yet illustrate the same fundamental ideas.

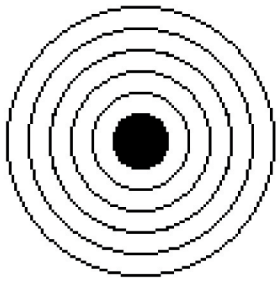

Figure 1: Picture corresponding to an object of a `RingedTarget` class.

For our example we will design a class that represents a target like that shown in Figure 1. A target consists of a small filled oval (the bullseye) and a series of concentric rings. A recursive description of a target is as follows. If the radius of the target is 15 pixels or smaller, the target simply consists of a bullseye. Otherwise the target consists of a framed oval with the given radius and then a smaller

[1]Introductory courses using functional languages, on the other hand, typically cover built-in recursively defined lists early.

target with the same center, but a radius 8 units smaller than before.

We can define classes representing targets that will have exactly this structure. The base class, `BullsEye`, will create the bullseye only, while the recursive class, `RingedTarget`, will represent targets with one or more circles around the bullseye.

We wish to define targets that support a `move` method and a `contains` method. Thus we define an interface with those methods:

```
public interface TargetInterface {
   // move the target by dx in x direction and
   // dy in y direction
   void move(double dx, double dy);

   // return whether the target contains pt
   boolean contains(Location pt);
}
```

Both classes to be defined will implement this interface. We use Java and emphasize interfaces over inheritance, but in courses that use other languages or emphasize inheritance over interfaces, a fully abstract class could be used instead.

Defining a `BullsEye` class implementing this interface is easy:

```
public class BullsEye implements TargetInterface {

   private FilledOval centerCircle; // bullseye

   // bullseye centered at pt with radius
   public BullsEye(Location pt,
                   double radius,
                   DrawingCanvas canvas) {
      centerCircle = new FilledOval(
                  pt.getX() - radius,
                  pt.getY() - radius, 2 * radius,
                  2 * radius, canvas);
   }

   // move the target by dx in x direction and
   // dy in y direction
   public void move(double dx, double dy) {
      centerCircle.move(dx,dy);
   }

   // return whether the target contains pt
   public boolean contains(Location pt) {
      return centerCircle.contains(pt);
   }
}
```

The class above uses the `objectdraw` graphics library, but the intent should be clear enough. The constructor draws a filled oval, while the `move` method moves the oval and the `contains` method determines whether the oval contains the point. As usual the base case of a recursive structure is very straightforward, and neither the constructor nor either method uses recursion.

The recursive class representing targets with one or more rings is more interesting.

```
public class RingedTarget
                   implements TargetInterface {
   // outer ring of target
   private FramedOval outer;
   // rest of target
   private TargetInterface rest;

   // Create target centered at pt with radius
   public RingedTarget(Location pt, double radius,
                       DrawingCanvas canvas) {
      // Create and center outer ring
      outer = new FramedOval(pt.getX() - radius,
                  pt.getY() - radius, 2 * radius,
                  2 * radius, canvas);
      radius = radius - 8;
      if (radius > 15) {
         rest = new RingedTarget(pt, radius,
                                 canvas);
      } else {
         rest = new BullsEye(pt, radius, canvas);
      }
   }

   // move the target by dx in x direction
   // and dy in y direction
   public void move(double dx, double dy) {
      outer.move(dx,dy);
      rest.move(dx,dy);
   }

   // return whether the target contains pt
   public boolean contains(Location pt) {
      return outer.contains(pt);
   }
}
```

The constructor creates an outer ring and then, depending on the size, sets `rest` to be either a new smaller `RingedTarget` inside the outer ring or a new `BullsEye`. The `move` method moves the outer ring and then moves the `rest`. The `contains` method, on the other hand, is not recursive and simply determines if the outer ring contains the location passed in.

While this example introduces the general use of recursion, it also reinforces other important concepts. Interfaces are being used in an essential way, as the value of `rest` can either be a `BullsEye` object or a `RingedTarget` object.

Even more important is how this code illustrates the importance of dynamic method dispatch. In the `move` method, the message send of `move` to `rest` is an excellent example. If the value of `rest` is a `BullsEye` object, then the `move` method of that class moves the `centerCircle`. If the value of `rest` is a `RingedTarget` object, then the `move` method will move its outer ring and then send the `move` message to its `rest` instance variable (which is of course different from the `rest` instance variable of the original receiver).

As noted earlier, the example above is a thinly veiled example of a recursively defined list. We provide students with several other examples of recursive structures that all fit the same pattern (e.g., lists of URLs, scribbles as lists of line segments) as well as one or more examples of somewhat different structures (e.g., fractal drawings of snowflakes or broccoli), each time keeping a focus on the general recursive structure.

4. WHY STRUCTURAL RECURSION?

Most textbooks and instructors introduce recursion via methods in which the recursion is based on integers. For example, one can define recursive algorithms for exponentiation, binary search, or quicksort, where the recursion is based on the number of elements in the slice of the array left to be processed.

While this is fine as an approach to recursion (and examples like binary search and quicksort can be important examples of the use of recursion), we argue that this sort of example is less intuitive to students and does not highlight the important object-oriented features that are brought out by the use of structural recursion. Let's address each of these individually.

Students find it hard to conceptualize recursive functions and procedures. Recursion works by starting one computation, and then interrupting that computation to perform one or more computations using the exact same set of instructions and instances of the same variables, eventually returning to complete the original computation. Instructors know that different copies of parameters and local variables are kept with the stack of activation records corresponding to the various recursive calls. The notion of an activation record, however, is new to the students and difficult for them to grasp because they are allocated implicitly and never explicitly manipulated by the program. While instructors may try to make the underlying execution model more concrete using exercises in which one student begins executing the original computation and calls on others to perform the recursive calls, the notion of procedural recursion resulting in new invocations of procedures is difficult for many students to grasp.

On the other hand, structural recursion is quite concrete. In a course like ours that stresses object-oriented programming, students will have already become familiar with the idea that it is possible to create multiple objects of a single class. Unlike activation records, the creation of these objects is done explicitly in their code. Moreover, they understand that each of these objects has its own copies of the instance variables declared by the class. This knowledge is developed through their programming experience before the subject of recursion is introduced. For example, in the assignment described above in which our students implement a program that lets them drag two magnets around the screen, they know that each magnet must have its own copies of the instance variables that keep track of its position. This is reinforced concretely by the fact that when they invoke the move method of one magnet the other does not move.

This knowledge can be easily exploited when explaining recursion. When they see that the `move` message is sent to a particular object from class `RingedTarget`, one can draw a picture to illustrate the instance variables of the original target and the objects to which they refer. When the recursive call is made to `rest`, the slightly smaller target, another picture can be drawn illustrating the separate instance variables of that object. This can continue all the way down to the base case. These objects are familiar to the students and more concrete than are activations of functions.

We remarked earlier on the fact that using data structures defined by structural recursion highlights and reinforces important concepts in object-oriented programming such as the use of interfaces (subtyping) and dynamic method dispatch. Unfortunately, methods defined based on recursion on the integers illustrate none of these principles. No interfaces are relevant, and the decision as to which code is to be executed is based on a conditional statement rather than on dynamic method dispatch. A binary search or quicksort in an object-oriented language looks extremely similar to one in a procedural language. In fact, many texts illustrate these with static methods, emphasizing their independence from object-oriented concepts.

Another advantage of using structural recursion is that students see these data structures representing collections of objects where the details are hidden inside classes. When we later teach students arrays as an alternative way to hold collections, they are much more likely to see the reasons for hiding the arrays as instance variables in classes that export the more natural methods associated with the data structure. We often show one data structure implemented by structural recursion and another version using arrays. The public methods with their signatures in each are the same, with the differences in implementation hidden inside of the objects. At that point we can discuss how both produce the same results, but with slightly different performance characteristics.

5. EVALUATION

There have been noticeable differences in student programs written in the second half of our course after we moved up the presentation of recursion and emphasized structural recursion over recursion on integers. For example, very few students now resist encapsulating arrays inside classes. Because of their experience with recursive data structures, they now see this as the normal way of handling data structures.

Recall our earlier example of the Simon game, and a class `Song` representing the sequence of notes generated so far. After the introduction to structural recursion, students are much more likely to create a class with an array as an instance variable and where the methods are not tightly associated with array operations. Instead they include methods like `addNote()`, `atEnd()`, `makeNewSong()`, `play()`, etc. As a result of this superior organization, they find this program easier to write and debug.

In checking to find quantifiable evidence of the impact of this change in student perceptions of our course, we examined student course evaluations, in particular the summary scores for course difficulty. Of course there are many variables that can contribute to student perceptions of course difficulty. To adjust for some of these, we compared two offerings of the course that were taught by the same lead instructor in a similar semester. (We see somewhat different student populations between the fall and spring semesters.)

In the spring of 2000, a semester in which arrays were taught well before recursion, students reported an average level of difficulty for the course of 4.0 out of 5, where 1 is the lowest difficulty and 5 is the highest. In the spring of 2002, where recursion was taught before arrays, students reported an average level of difficulty of 3.0 out of 5. The first score was in the highest quintile range for courses in the college, while the latter falls into the second lowest quintile range. We note that most of the lab assignments were essentially the same between these two course offerings.

While there are always minor differences between offerings of the course and indeed the population of students, the changes in student reported level of difficulty suggests

that students perceived the course to be easier when recursion was moved earlier. The earlier version of the course introduced integer-based recursion before structural recursion, though the associated lab assignment, both before and after the earlier introduction of recursion, involved designing a recursive class representing a fractal-like picture. Later versions of the course have also included a second recursion lab involving writing a program to handle lists of scribbles.

6. RELATED WORK

There have been many papers on teaching recursion (for example, Wu et al [8]), but only a few have focused on structural recursion. Structural recursion is very common in functional programming languages. An example is the Felleisen et al text [4], which uses Scheme and emphasizes strongly the design of programs based on the structure of the data. Another example is the paper by Henderson and Romero [6], which discusses teaching structural induction with ML.

The text [5] by Felleisen and Friedman breaks from the more usual approaches to Java by developing immutable recursive lists (representing pizzas) from the very beginning. The style used in the book is very similar to that used in functional languages. We do not advocate going as far as those authors, as we believe it is important for students to understand mutable variables and loops, but we do believe that introducing recursion before arrays can have great benefits.

Aside from that text, there has not been as much emphasis on structural recursion early in procedural or object-oriented languages. Our examination of several recent Java-based CS 1 books has shown that most text books either introduce recursion as an optional section in one or more chapters or relegate it to a very late chapter where it is unlikely to be covered. Moreover, virtually all of the examples of recursion presented in these texts involve procedural recursion on integers rather than the structural recursion we are advocating here.

One of the few examples of papers discussing structural recursion we were able to find is Nguyen and Wong's paper [7], which discusses using the visitor pattern on recursively defined data structures, a more advanced topic than we have discussed here.

7. CONCLUSIONS

In this paper, we have argued that the introduction of structural recursion before the presentation of arrays helps solidify students' understanding of object-oriented concepts. It also increases students' understanding of why other data structures, such as arrays, also need to be encapsulated in classes that provide operations more naturally associated with the objects represented by the classes. Moreover, we argued that it is pedagogically easier to explain – and for students to understand – structural recursion than recursion based on integer values. Finally, presenting structural recursion as described here reinforces the use and understanding of interfaces and dynamic method dispatch.

We have been writing a text [3] based on our approach to teaching programming in a CS 1 course. Instructors who are interested in seeing more of the details of our approach to presenting recursion will find them in the draft chapters of the text that are available at:

`http://eventfuljava.cs.williams.edu/`

The text is now scheduled to be published by Prentice Hall in early 2005.

8. REFERENCES

[1] K. B. Bruce, A. Danyluk, and T. Murtagh. Event-driven programming can be simple enough for CS 1. In *Proceedings of the 2001 ACM ITiCSE Conference*, pages 1–4, 2001.

[2] K. B. Bruce, A. Danyluk, and T. Murtagh. A library to support a graphics-based object-first approach to CS 1. In *Proceedings of the Thirty-Second ACM SIGCSE Symposium*, pages 6–10, 2001.

[3] K. B. Bruce, A. Danyluk, and T. Murtagh. *Java: An eventful approach*. Prentice Hall, 2004.

[4] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[5] M. Felleisen and D. P. Friedman. *A little Java, a few patterns*. MIT Press, 1997.

[6] P. B. Henderson and F. J. Romero. Teaching recursion as a problem-solving tool using standard ML. In *Proceedings of The Twentieth SIGCSE Technical Symposium on Computer Science Education*, pages 27–31. ACM Press, 1989.

[7] D. Nguyen and S. B. Wong. Patterns for decoupling data structures and algorithms. In *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, pages 87–91. ACM Press, 1999.

[8] C.-C. Wu, N. Dale, and L. J. Bethel. Conceptual models and cognitive learning styles in teaching recursion. In *Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*, pages 292–296. ACM Press, 1998.