A TOP-DOWN APPROACH TO TEACHING PROGRAMMING

Margaret M. Reek
Department of Computer Science
Rochester Institute of Technology
102 Lomb Memorial Drive
Rochester, NY 14623
mmr@cs.rit.edu

Abstract

Programming is traditionally taught using a bottom-up approach, where details of syntax and implementation of data structures are the predominant concepts. The top-down approach proposed focuses instead on understanding the abstractions represented by the classical data structures without regard to their physical implementation. Only after the students are comfortable with the behavior and applications of the major data structures do they learn about about their implementations or the basic data types like arrays and pointers that are used. This paper discusses the benefits of such an approach and how it is being used in a Computer Science curriculum.

1. What do we mean by a top-down approach?

In this context, "top-down" refers to teaching programming with an initial emphasis on the use of pre-existing components, deferring low-level implementation issues until later in the student's studies. These components may include the classical abstract data types such as stacks, queues, lists, and trees, or sections of an application program such as an ATM machine or a card game.

We choose to focus on the abstractions in the belief that they are the more important concepts. Our experience has been that implementation issues distract the students to the point that they do not really understand the abstractions, particularly with the classical data structures.

Partial support for this work has been provided by the National Science Foundation's Instrumentation and Laboratory Improvement Program, award #DUE-9451123

SIGCSE '95 3/95 Nashville, TN USA

2. Problems we saw in a bottom-up approach

Programming is typically taught using a bottom-up approach. Starting with simple statements, we go on to Boolean expressions and conditional statements, procedures, and finally arrays and records. Only then are the abstract data structures such as stacks, queues, trees addressed. We show students what a stack is and how it is implemented, first with an array and later with pointers and dynamically allocated storage; this is the approach taken in many texts [1,2,4,13].

At RIT, we have been using this same basic approach since the inception of the department in 1972. As abstract data types gained importance, we tried to teach our students to think about them in more general terms. However, our bottom-up approach thwarted these efforts. All of the students' previous education concentrated primarily on implementation details, so some had a hard time thinking about the abstract behavior being implemented. They perceived a *stack* as "an array that is manipulated in this way". The students found it difficult to see the abstract behavior through the forest of implementation details, even when it was pointed out. We then tried starting with the abstract concept and showing examples. This did not work well, because the examples were immediately followed by a discussion of how to build one, and the abstract behavior once again became tied in their minds to a specific implementation. Our approach of examining a data structure like a stack abstractly, seeing some uses for it, and then seeing how it could be implemented failed because it left the student with the idea that the implementation was the important part. Students' minds are sometimes like a stack: the most recent item put in is all that is accessible!

We tested the student's ability to work with abstractions by asking questions such as this on tests:

> Write a procedure named *BottomStack* that returns the bottom element of a stack while leaving the contents of the stack unchanged. You **MUST** make use of standard stack operations, such as those found in the attached listing of a prototypical stack library named *StackLib* - you will receive NO POINTS for this problem if you access the stack directly.

A significant portion of the students couldn't solve the problem without accessing the stack directly. This was

after discussions about and examples of similar problems, and debating the benefits of using the interface rather than accessing the data structure directly, as the solution would then be independent of the actual implementation.

The students' thinking of abstract structures in terms on a concrete implementation made it difficult to talk about general issues, such as when a particular structure is appropriate and the tradeoffs in using one structure or another to solve a problem. The students couldn't find the forest because they couldn't see past the leaves, much less the trees.

The problem isn't that the students are stupid, but rather that at age eighteen their thinking maturity is still at the concrete level. They are much more comfortable with concrete than abstract thinking, and the bottom-up approach makes it easier for them to fall back on their old habits rather than developing new ones. It is likely they will feel uncomfortable for a while with the top-down approach as it forces them to deal with the abstraction so much of the time. Immersing them in this way of thinking should get them over this hurdle faster.

Another problem with the bottom-up approach is that students with experience see the same things they already know, and it reinforces a tendency in some students to think they know it "all". When we get to new material a few weeks into the course, they've already stopped coming to class because they think there is nothing new to learn; these students have problems later because they've missed so much. Also, many find the material dull because it is difficult to find interesting problems for them to solve with the limited set of skills they have.

3. Benefits of a top-down approach

In the fall of 1994 we are starting to teaching programming using a top-down approach. We are making this change because we believe that it will solve the major problem of the bottom-up approach, as well as having other benefits. At this writing these are anticipated benefits; the experiences section will describe our first results.

The first benefit of this approach is that we can discuss the classic data structures such as stacks, queues, lists and trees much earlier because we are not discussing implementations. The students will be using predefined routines from libraries, so the only basics they need to know are how to write a sequence of procedure calls and pass parameters. They don't need to know the details of how procedures are declared, or the differences between every parameter passing technique, just enough to allow them to use something that already exists to solve a new problem. We teach stacks and lists in the fifth week of the first term, even before conditional statements are covered. If we were not spending a week on testing, we would reach stacks and lists even sooner.

A benefit of focusing on the use of components rather than their implementation is that students can do "interesting" things much earlier than they can in the traditional approach [7]. With knowledge of the abstract behavior of the classical data structures, students can solve moderately complex problems because they can focus on using the right tools to implement a solution rather than on implementing the tools themselves. Our students will work with a fairly complex card game, complete with a graphical interface, even before we teach any data structures; all they need to know to see the high level logic is how to identify which library components are needed and how to call them in the appropriate order.

We believe that this approach will help students to think of programs as collections of large components interacting with each other rather than a single very long sequence of basic instructions. Only when they are comfortable with this thinking do we discuss the implementations of the tools they have been using; this will occur at the beginning of the sophomore year. Some texts have some of the flavor of this approach [5,6,13] but most still deal with the implementations in the same course and text. We are pushing the conceptual introduction and the implementation much farther apart to separate the two in the student's mind.

We recognize that it is important for Computer Science students to be able to implement the tools they will be using, but by delaying this for so long we hope to get them used to using existing tools to solve problems. We believe that this will prevent them from getting the idea that "writing a program" always means implementing every bit of every part of it themselves.

This approach will also get students used to the idea they don't have to know *how* something is implemented to use it, which is a crucial step in using libraries and reusing components. Also using existing tools without knowing how they are implemented will make it painfully clear that good documentation and expressive names are crucial, not just something you do to get a good grade. If these are missing, it is not possible to use an existing tool without seeing the implementation. We will use examples that demonstrate both good and bad style and have the students work with them to reinforce this point.

4. Language support needed

The top-down approach can be taken, to some extent, with any language that allows the definition of the interface of a module to be separated from its implementation; either a procedural or an object-oriented language may be used. However, some languages will be better than others. A language like Modula-2 provides a reasonable degree of abstraction, but not generic types. We used Modula-2 for many years, but its lack of genericity

was limiting. For example, after discussing the abstract behavior of stacks, we were unable to implement a truly generic stack module; the language required it to be declared as a "stack of some specific type". The best we could do was to abstract the data type by putting the details of the individual element and some basic operations on it into a separate module, but still could not use one stack module in a single application to handle both a stack of integers and a stack of records. Languages that support generic types would alleviate these kinds of problems.

A library of existing components with a consistent interface is crucial; this can either be purchased or developed in-house. Consistency makes it easy to teach students how to search the library for the needed components and features.

5. How we are using this in our curriculum

We are using this approach in our first two years for Computer Science and related majors starting in Fall 1994. Stacks and lists are introduced in the first course, and queues and trees in the second course; these are 10 week courses. It is not until the start of the sophomore year that we look at concrete implementations of these using arrays and pointers. By that time the students will have ample experience using these structures and should be completely comfortable with them. They will have also learned a second language by then, which will help reinforce the idea that the abstract type is more important than the details of how it is implemented in one language or another.

We are also switching to a strong software engineering approach using an object-oriented environment for our new curriculum. We are using Eiffel [9] in the first three courses, which supports true generics and comes with a very large library of components. This language facilitates our top-down approach, though other languages could also be used.

6. Experiences to date

At the time of this writing, the first course has been offered once. The results so far have been promising. The students seem to have a fundamental grasp of the essence of the data structures, to the extent that they have figured out many things on their own without us telling them. For example, they intuitively know that a stack is something that can be used to reverse the order of data. This sounds obvious to us, but many students in previous years couldn't grasp that concept even though we explicitly told them. Now, when presented with problems that have them choose the most appropriate data structure, most students are able to do it. A further test of this will come in the second course where many more data structures will be presented.

We were concerned about the growning disparity between incoming students with programming experience and those without, and had hoped to "level the playing field" through the top-down and object-oriented approaches in our curriculum. This has been a bit disappointing. Both groups of students seemed to grasp the basic concepts equally well. However the non-programmers typically had a harder time expressing themselves in terms of the designing control structures. The problem was in the our implementation, not the approach. We spent a lot of time on high level design examples, but what the students really needed to see at least once was the entire process from specification to code. In the future, we will spend more time in class on complete examples from design through implementation and testing. A series of exercises similar to the "gateway labs" used at SUNY Geneseo [3] would also be helpful; we've developed several of these using Mosaic, but more are needed.

We are finding that attendance in both lecture and lab is much higher than in the past. The experienced students realize right away that they don't know everything so they come to class. We can also do more interesting assignments early in the term, which tends to capture all the students' interest. The withdrawal rate from the course is lower than in the past, even though the overall course GPA is roughly equivalent. The percentage of students continuing on to the next course is higher than in the past. It would appear that we are able to capture and hold their interest better than before; only time will tell if what we are seeing is due the approach itself or is merely the "Hawthorne effect".

We are finding that we can present the abstract data structures and their uses more easily now because students are used to the library notation and seeing the the class interface but not the coding details. A pleasant surprise was that the students didn't seem to mind not being shown how things were done and virtually no one asked questions about it. They didn't even seem tempted to look at the code, although they had ample opportunity to do so. We kept reinforcing the notion that the interface was the important part: we don't really care how it is implemented, as long as the job gets done as advertised. They are comfortable with the idea that you don't have to write every piece of code yourself, nor do you have to know how something was implemented to use it. This reinforces the notion of "design by contract" which we have been stressing.

Lastly, we are really pleased with our choice of first language. Eiffel and the environment we are using [8] are extremely well suited to our goals. The syntax diagrams for the entire language fit easily on a double sided sheet of paper (yes, in type large enough to read), yet its support for design by contract and reuse is superb. A very complete set of libraries [10] is provided. Their

consistent presentation and naming convention makes it easy for students to learn to read the interfaces to find the features they need. For example, on the final exam we gave them an interface to a library class that they were unfamiliar with and asked them to use it to solve a small problem. Even though they were unfamiliar with this particular class, the vast majority were able to solve the problem satisfactorily. They also recognize the similarities amongst initially disparate data structures more easily. For example, many realize that a list and a file are essentially similar and should have many features in common. Because the libraries we use have a consistent structure the students see that this is indeed the case.

The biggest problem we've had in this venture is lack of appropriate textbook support for both our approach and the language. We use *Designing Object-Oriented Software* [14] as a language independent book on software development. It is a very nice treatment of the subject, but doesn't have any of the data structure information the students would find helpful. The definitive text on Eiffel is *Eiffel: The Language* [9] but it is not appropriate for an introductory programmer. We are currently using *Eiffel: An Introduction* [12] but it doesn't have enough detail or complete examples to be adequate for beginning students. A number of new texts are about to be published and we hope to find one more appropriate for our needs.

7. Looking to the future

At this writing the second course is just about to start. We anticipate that the students will continue to grasp the basic concepts, and we will try to remediate some of the mistakes from the first course. The first course is being offered again this term, and we will be fixing the problems we observed the first time around. By the time of the conference, our second quarter will be completed and we can present results of this next phase.

We are particularly interested in seeing how well the students are able to retain the abstract concepts in the third course, where we start the switch from Eiffel to C++. Of major concern are the more involved inheritance concepts that are supported so cleanly in Eiffel and less elegantly in C++. It will also be interesting to see whether the students who couldn't wait to learn C++ because is it a "real" language will be equally vocal about wanting to stay with Eiffel because of its simple yet powerful support for object-oriented design. We'll let you know next year.

8. References

(1) Bergin, J. *Data Abstraction: The Object-oriented Approach Using C++*. McGraw-Hill Inc. 1994.

(2) Budd, T. *Classical Data Structures in C++*. Addison-Wesley Publishing Company. 1994.

(3) Cowley, B; Scragg, G.; Baldwin, D. "Gateway Laboratories: integrated, interactive learning modules" *SIGCSE Bulletin*, Vol. 25 No. 1, pp 180-183 1993.

(4) Dale, N; Lilly, S. *Pascal Plus Data Structures, Algorithms, and Advanced Programming* D.C. Heath and Company. 1991

(5) Decker, R; Hirshfield, S. *The Object Concept: AN Introduction to Programming Using C++*. PWS Publishing Inc. 1994.

(6) Headington, M; Riley, D. *Data Abstraction and Structures using C++*. D.C. Heath and Company. 1994.

(7) Hilburn, T. "A Top-Down Approach to Teaching an Introductory Computer Science Course." *SIGCSE Bulletin*, Vol. 25 No. 1, pp 58-62. 1993.

(8) Meyer, B. *Eiffel: The Environment.* Prentice Hall. 1994

(9) Meyer, B. *Eiffel: The Language.* Prentice Hall. 1992

(10) Meyer, B. *Eiffel: The Libraries.* Prentice Hall. 1994

(11) Nyhoff, L; Leestma, S. *Data Structures and Program Design in Modula-2.* Macmillan Publishing Company. 1990.

(12) Switzer, R. *Eiffel: An Introduction* Prentice Hall. 1993.

(13) Tucker, A; Bradley, WJ; Cupper, R; Epstein, R. *Fundamentals of Computing II: Abstraction, Data Structures and Large Software System.* McGraw-Hill Inc. 1993.

(14) Wirfs-Brock, R.; Wilkerson, B.; Wiener, L. *Designing Object-Oriented Software.* Prentice Hall. 1990.