TOWARD A FIRST COURSE BASED ON OBJECT-ORIENTED PATTERNS

Eugene Wallingford
Department of Computer Science
University of Northern Iowa
Cedar Falls, Iowa 50614-0507
wallingf@cs.uni.edu

1 INTRODUCTION

Many different approaches to teaching introductory courses in computer science have been explored in the last few years. Of these, most aim to resolve the conundrum of how to teach abstraction skills in the face of student difficulties with learning to write programs. CS1 courses often carry the added burden of trying to present a more accurate account of what computer science is like. To this end, special emphasis is now placed on topics such as code reuse. Even more new problems arise when an object-oriented approach is adopted for use in introductory classes. What are the appropriate abstractions to teach? What are the best vehicles for teaching them?

This paper advocates an approach to teaching object-oriented (OO) introductory computer science courses that is based on programming *patterns*. Section 2 examines the problem of abstraction versus detail in introductory courses and motivates the use of patterns. In Section 3, the pattern approach is described. That section outlines the approach as it has evolved over three years of procedural programming instruction and begins to develop the idea of patterns for teaching OO-based courses. Section 4 relates this approach to other work on teaching introductory courses. Finally, Section 5 presents future plans for developing and refining the approach.

2 THE PROBLEM OF ABSTRACTION

The goals for most first courses in computing are difficult to achieve. Traditionally, CS1 aimed to teach the programming skills necessary to complete a program in computer science. Increasingly, this course is also being used to introduce students to other topics such as:

- "breadth" issues of computing as a discipline
- software engineering issues such as reuse

In most cases, these new goals create a demand for teaching at a higher level of abstraction in CS1. This demand is significant in its own right, but it is made even more problematic by the desire to still have students develop a reasonable level of programming skill.

The conflict between abstractions and the details of programming are present even in courses that focus only on teaching programming skills. Such courses have typically presented a top-down approach to program development. Yet Rist [15] found that students still tended to reason backward from the program goal through program code to a solution plan. This tendency arose because they lacked appropriate high-level schema for solving the problem. However, when students did have relevant schema available, Rist found that students could and did reason forward from plan to code. One of his conclusions was that novice programmers were not capable of applying general plans in the development of code but rather needed access to problem-specific plans.

One way around this obstacle is to teach suitable plans as abstractions. Yet students in CS1 usually have a great deal of difficulty just learning to use a programming language. The details of the language are foreign enough that learning to express a solution to a computational problem in it consumes an inordinate amount of their attention. Even when using a language with a simple syntax, such as Scheme or Smalltalk, novice programmers often do not have sufficient background for translating high-level actions into programming language statements. Requiring that students also learn abstract programming plans and other high-level abstractions makes their task even more burdensome. CS1 instructors face the dilemma of forging a middle ground between these seemingly incompatible desires.

In an effort to teach abstractions more directly, a growing number of computer science departments have begun to adopt an object-oriented approach in CS1. This trend seeks to realize many of the same benefits in computer science education that the OO paradigm brings to software engineering. The benefits of different OO approaches

in CS1 have been recounted in recent years by several educators [8,14]. Among these are that OO programming:

- encourages well-structured programs
- encourages reuse of code and design
- allows early use of complex objects

In CS1, these features also provide the advantage of supporting a better introduction to computer science, through more realistic modeling experiences with larger and more interesting tasks.

Despite these potential benefits, computer scientists know relatively little about how to teach this paradigm in introductory courses. What we do know indicates the need for care in instruction. Detienne [4] has shown that students have difficulties switching from another paradigm to an OO approach. This situation may result from students' inability to overcome a centralized mindset when designing code, as reported by Guzdial [7]. Additionally, popular OO programming languages are not always appropriate for teaching novices [8]. These factors conspire to make learning problematic. Student attention is focused on an discomfiting language syntax, which distracts them from the already difficult task of learning a new problem-solving paradigm.

If students are to reason in a top-down fashion, then they must either be provided with relevant schema for solving problems or given plenty of time to develop their own through practice. Traditionally, CS1 students have been left to construct their own schema implicitly, through variety in programming exercises and lecture examples. The results of this technique are not always satisfactory, due to the wide range of possible schema that can be learned. Students often end up creating schema that are too narrow, stylistically displeasing, or just faulty. Remediating these inappropriate schema takes time that might otherwise be used for teaching other abstraction and programming skills. Rist's research inspires a possible solution: Teach programming schema that are known to produce good programs.

3 PATTERNS

With this motivation, I have been refining an approach to introductory courses that is based on *programming patterns*. Patterns have long been viewed as an important part of how we understand and shape the world around us. In the context of design, patterns [2,9] are recurring solutions to the problems that confront designers. Each pattern encapsulates a well-defined problem and a standard solution. This solution consists of both a design and a description of how to implement it. By thus intertwining the problem, the solution, and the implementation, patterns allow the designer to refine them concurrently during system development. The use of patterns provides a new level of abstraction at which analysis, design, and implementation can be done.

In the context of software development, patterns embody abstractions that are used to write and understand software. A pattern approach offers the novice programmer a new kind of tool. Rather than viewing programming language statements as the building blocks out of which to construct programs, students can be taught to use patterns as the basic unit of analysis, design, and programming. These patterns provide a mapping from a type of problem to an effective algorithm and an effective implementation in code. In this way, small piecework that the students would otherwise have to continually redo is standardized into a larger unit that can be reused in various contexts.

3.1 Patterns in Procedural Programming

I have used patterns to teach procedural programming for the last three years. Figure 1 lists the procedural patterns used in these courses. This set provides a framework sufficient for teaching all of the material covered in an introductory programming course.

Input-Process-Output

Guarded Action
Alternative Selection

Process All Items in a Collection
    Counting
    Accumulating
    Finding Maximum/Minimum

Linear Search through a Collection
    Interactive Data Validation

Figure 1: Procedural Patterns for Novices

Figure 2 gives an example of a pattern used in our old Pascal-based CS1 course. The counting pattern is the simplest repetition pattern that I cover and serves as an introduction to repetition structures. Three elements comprise each pattern description: a statement of the general problem, an algorithm for solving the problem, and a code pattern that solves the problem. In this approach, students analyze a problem by first trying to identify general problem features that match one of the patterns in their "catalog." They then stylize the code pattern to match the details of the problem. The commented steps of the code always need to be accounted for with problem-specific code, but the rest of the pattern can also be modified if necessary.

For example, if a problem asked for the number of words read from standard input, the student would simply replace the comments with code to read values from standard input. If a problem asked for a count of the number of words beginning with the letter 'h', the student would also modify the incrementing of count to only be done under the

proper condition (using a *guarded action*, a selection pattern that they have already learned). If the problem asked for similar counts on a collection stored in an array, the get and loop steps would be modified to use a for statement over the array's range of indices. A central element of the course is exploring when and how to modify code patterns for specific applications.

The course is organized around a number of prototypical patterns and a broad selection of different kinds of problems. These problems allow the student to gain experience recognizing the presence of a pattern, applying a pattern with minimal modifications, and applying patterns in contexts that call for more significant modification. Once a number of patterns have been introduced, we cover more complex data types, such as arrays and files, and examine the kinds of modifications that each typically requires. Students begin to see patterns in the modifications themselves, which reinforces their understanding of the data type.

Pattern: Counting

Problem: Need to count the number of items in a collection of values

Algorithm:
```
Initialize counter to 0
While there are more items,
        Process the item
        Increment the counter
```

Code:
```
count := 0;
/* get the first value */
while (value <> STOPPER)
{
  /* process the value */
  count := count + 1;
  /* get the next value */
}
```

Figure 2: The Counting Pattern

3.2 Evaluation of the Experience

The pattern approach has evolved through seven semesters of use in introductory courses, both in CS1 and in standard procedural programming courses. In addition to Pascal, it has been used in courses teaching BASIC and C++. I have done no formal analysis of the effects of the approach on student performance and learning, but I can offer several comments on my perception of its effects.

First, many students do seem to benefit from the use of patterns as the basic units for writing programs. Weaker students now rarely face the daunting task of creating programs on an empty editor screen, one line at a time. By recognizing an appropriate pattern in a problem, they can immediately begin working with a "chunk" of meaningful code. This effect has been most noticeable on exams, where students are under time constraints. Second, the approach has encouraged reuse of code and algorithms from problem to problem. Students have commented that they can view the patterns as tools, so that by the end of the course they are well-equipped to handle nearly any problem that they encounter.

However, concerns about the approach have arisen. Some students feel that the use of patterns limits their ability to create unique solutions of their own. Not all are persuaded by claims that patterns *channel* their creativity toward the most interesting elements of a solution. In the same vein, though, I have been concerned that the use of patterns might inhibit the better students' development of their own problem-solving schema. By exposing students to a wide enough range of problems, this difficulty may be avoided. The more that they have to modify and combine patterns, the more that they will have an opportunity to develop their own. Still, monitoring student progress in future courses may provide a more concrete answer.

The results of the pattern approach have been encouraging enough that a group of faculty in our department is now working on a project, funded by the National Science Foundation [5], to develop a full set of curricular materials based on the approach. These materials will include a suggested set of patterns for use in a first course in programming. The project will also evaluate the approach further by using it in courses teaching C and COBOL.

3.3 Patterns in OO Programming

Since the fall of 1994, our department has used an object-oriented approach in CS1 and CS2. As a result, I have begun to consider how I might apply the pattern approach to an OO first course. An object-oriented pattern is typically a small set of classes, usually two or three in number, that is frequently useful in program development. The key to an OO pattern is the relationships between the objects that comprise it. As with procedural patterns, OO patterns provide a vocabulary for building systems, one that is more helpful than individual classes.

A great deal of the research on patterns originated and still resides in the OO community. However, not all of that work is directly relevant to my goals. First, that work targets primarily more advanced software practitioners. Second, it tends to emphasize design issues [6,9], whereas a first course must provide a significant amount of support for novice *programmers*. Thus the patterns that we use in such a course will tend to be more concrete and low-level compared to the ones in the literature on patterns.

I have begun to identify a small set of OO patterns for use in our new CS1 course. For teaching purposes, the selected patterns must be concrete enough to provide meaningful support for writing implementation code. As noted above, the concerns of novices generally focus on

the generation of working code. This means that some patterns will involve single objects, providing students with concrete support for coding objects that play particular roles. Also, while our goal in CS1 is not primarily to develop design skills, the patterns introduced there should be general enough to find application in the student's later experience. Figure 3 gives brief descriptions of the patterns I have chosen so far.

Figure 4 offers an abbreviated characterization of the view pattern in C++. One of the simpler OO patterns, views will serve as an introduction to the composition of objects. Three elements comprise each pattern write-up: a statement of the general problem, a description of the objects that solve the problem and their relationships, and a code skeleton that implements the description. As with procedural patterns, the programmer must furnish problem-specific code for the commented steps of the pattern. This task will be supported by single-object patterns covered earlier in the course.

State Maintain a body of data and provide suitable access to it by other objects and human users.

View Decouple an object's state from its presentation to other objects and human users.

Decorator Attach additional functionality to an object without modifying the class.

Figure 3: Possible OO Patterns for Novices

OO patterns differ from procedural patterns in that they emphasize object roles and relationships rather than relationships among actions. But the process for using OO patterns in teaching will remain much the same. The student will examine existing code in order to recognize patterns in existing code. They will analyze new problems by first identifying problem features that match one of the patterns in their "catalog" and then stylizing the code pattern to match the details of the problem. And much emphasis will be placed on adapting the features of patterns to the details of specific types of problem.

My plan is to develop approximately six patterns that will serve to introduce OO analysis, design, and programming to novices. One of the key goals in this effort to help students overcome the centralized mindset that they bring to problem solving. As Guzdial [7] has reported, students tend to centralize control of processing in a single function or object, except when they are very familiar with the system of objects involved. OO patterns provide a mechanism for seeing solutions to problems in terms of distributed objects with distributed control. By providing students with numerous examples of good problem decomposition and with numerous opportunities to apply patterns, I hope to help students overcome the tendency toward centralized programs.

Pattern: View
Problem: Need to display an object in a way that is not intrinsic to the object

Objects: Model, which maintains state data
View, which contains the model

Code:

```
class model
{
  public:
    /* constructors */
    /* access functions */
  private:
    /* state data */
};

class view
{
  public:
    /* constructors */
    /* function to access model */
    /* display functions */
  private:
    model my_instance;
};
```

Figure 4: The View Pattern

4 RELATED WORK

The idea of using patterns to teach procedural programming is not new. Both Soloway's plans [16] and Linn and Clancey's templates [10] capture similar "chunks" of programming knowledge. Soloway proposed a number of useful plans and a method for decomposing problems that use them. Linn and Clancey expanded upon the idea of a plan to offer much more complete templates of programming practice. Yet their approach emphasizes case studies, not templates, as the basis for pedagogy. The pattern approach, however, differs from both approaches by making patterns the central focus of the course. It proposes (a) a methodology for teaching an entire first course and (b) a complete set of patterns for this purpose.

Recently, several educators have begun to explore the use of apprenticeship [1] approaches in introductory computer science instruction. In this model, students learn to program by first reading and modifying programs that have been written by experts [11,13]. Only after they have experience of this type do they write new code on their own. Such approaches can be used to teach either procedural or object-oriented programming. The use of patterns is quite compatible with an apprenticeship model.

Patterns provide a high-level vocabulary for studying and understanding expert code. Later, they provide a skeletal structure from which to write new code.

An exciting application of this idea specifically to OO programming involves the use of "frameworks" [3,12]. A framework is a body of reusable code, usually expressed as a set of classes. These classes are used as the basis for developing applications of a particular kind, such as a graphics programs or database programs. Students learn to program by extending the framework with problem-specific classes of their own, using inheritance. Well-designed frameworks provide the novice with objects that are powerful enough to be interesting yet simple enough to be understood. Thus students are able to learn programming in the context of a real application guided by expert code.

Again, the use of patterns is quite compatible with a framework approach. A framework provides examples of good design and programming, but it does not provide specific tools for understanding the classes in the framework or the relationships among them. How does one study a framework? Or extend it? Why is the code in the framework considered "good"? Patterns offer a vocabulary for answering these questions. By using patterns, an instructor can teach the fundamental concepts of OO programming through the content of specific problem-solving schema. One way to explore the relationship between frameworks and patterns would be to construct a framework specifically intended for use with a specific set of OO patterns, or to select a set of patterns to teach based on the relationships inherent in an existing framework.

5 FUTURE PLANS

Teaching introductory computer science courses is difficult partly because of the problem of abstraction: Students have a hard time operating at a high level of abstraction while learning the low-level skills of programming. The approach described here offers a way to temper this problem by teaching programming in terms of patterns. Patterns provide high-level building blocks that guide the student's use of language constructs. They also encourage the reuse of software components. This approach has been applied informally to teach procedural programming for three years, with some success. However, the trend toward teaching introductory courses using object-oriented techniques creates a need to develop a first course based on object-oriented patterns of design and programming.

This paper describes the rudiments of such a course, but much work remains. First, a full set of object patterns must be identified. Second, a course based on these patterns must be designed. The challenge of this task lies in discovering useful patterns that are:

- relevant to the student's future experiences
- simple enough for novices to understand
- provide sufficient programming support

Such a course must teach the desired object-oriented concepts while also preparing students to implement methods using simple procedural code. This may involve use of procedural patterns in some form. My goal is to be able to present experience in such an object-oriented first course based on patterns by March of 1996.

6 REFERENCES

1. Astrachan, Owen, and David Reed (1995). "AAA and CS1: An Applied Apprenticehip Approach to CS1," *SIGCSE Bulletin* 27(1):1-5.

2. Coad, Peter (1992). "Object-Oriented Patterns," *Communications of the ACM* 35(9):152-159.

3. Conner, D. Brookshire, David Niguidula, and Andries van Dam (1994). "Object-Oriented Programming: Getting Right at the Start," OOPSLA '94 Education Symposium.

4. Detienne, F. (1990). "Difficulties in Designing with an Object-Oriented Programming Language," INTERACT '90, Cambridge, England.

5. Pattern-Based Programming Instruction (1995). NSF Grant DUE-9455736.

6. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns.* Addison-Wesley.

7. Guzdial, Mark (1995). "Centralized Mindset: A Student Problem with Object-Oriented Programming," *SIGCSE Bulletin* 27(1):182-185.

8. Kolling, Michael, Bett Koch, and John Rosenberg (1995). "Requirements for a First Year Object-Oriented Teaching Language," *SIGCSE Bulletin* 27(1):173-177.

9. Lea, Doug (1994). "Christopher Alexander: An Introduction for Object-Oriented Designers," *ACM Software Engineering Notes*, January 1994.

10. Linn, Marcia C., and Michael J. Clancy (1992). "The Case for Case Studies of Programming Problems," *Communications of the ACM* 35(3):121-132.

11. Pattis, Richard E. (1993). "The 'Procedures Early' Approach in CS1: A Heresy," *SIGCSE Bulletin* 25(1):122-126.

12. Pattis, Richard E. (1995). "Teaching OOP in C++ to Novices by using an Artificial-Life Framework." Submitted to 1996 SIGCSE Technical Symposium.

13. Reek, Margaret M. (1995). "A Top-Down Approach to Teaching Programming," *SIGCSE Bulletin* 27(1):6-9.

14. Reid, Richard J. (1993). "The Object-Oriented Paradigm in CS1," *SIGCSE Bulletin* 25(1):265-269.

15. Rist, Robert S. (1989). "Schema Creation in Programming," *Cognitive Science* 13:389-414.

16. Soloway, Elliot (1986). "Learning to Program = Learning to Construct Mechanisms and Explanations," *Communications of the ACM* 29(9):850-858.