Pattern-based Programming Instruction*

J. Philip East, S. Rebecca Thomas, Eugene Wallingford, Walter Beck, Janet Drake
University of Northern Iowa/Marist College

Several years ago a group of our computer science faculty began seriously examining initial programming instruction. We discovered a shared perception that too many students don't write reasonable programs even after completing a semester course in programming. Others have noted the same may even be true after the second course (e.g., [1]). There is cause for concern as computer applications pervade our society, often in life-critical situations. Our students are producing some of those applications.

Typical programming instruction tries to both cover syntax and instill skill in algorithm development. Students are quickly writing their own programs after minimal instruction and an example or two. They write programs by putting statements together in ways that, as often as not, are poor if not outright incorrect. Instructors or graders provide some feedback to the students after several days. By that time instruction has moved on and another assignment has been assigned. Understandably, students pay little attention to the feedback and too often have little chance to apply it even if they did attend to it. As this happens repeatedly during the term, the end result is that too large a fraction of the students never actually produce a good program. Many introductory texts address top-down-design by admonishing students to break larger problems into smaller problems and by giving static examples that illustrate a very dynamic process. Students seem to ignore the examples and get no insight into how problems can be broken down. The texts illustrate an overwhelming concern with the "what" of instruction but almost no knowledge or insight into the "how".

An additional concern about programming instruction relates to the improvement of pedagogy. There seem to be few espoused principles for designing instruction. Astrachan and Reed [1] indicate most

---

This material is based upon work supported by the National Science Foundation under Grant No. DUE-9455736.

texts are organized according to language constructs rather than computer science concepts. Pattis [7] notes that in at least one case (procedures early), the conventional wisdom has "cramped the pedagogical discussion of what is the best sequence and placement of topics ..." There is indeed little discussion of the teaching of programming that relates to pedagogy and almost none that addresses how the process of learning might or should affect instruction.

Various suggestions for the improvement of programming instruction do explore something beyond language features. Clancy & Linn [3] advocate using case studies to more thoroughly explore programming problems, perhaps to provide deeper understanding. Astrachan and Reed [1] expect "the apprenticeship model ensures that by extending programs, and eventually developing them from scratch, good design skills are inculcated over time." (p. 1) Soloway [8] suggests that expert programmers think in terms of goals and plans that are canned solutions to problems. He further argues that stepwise refinement should occur "on the basis of problems that you have already solved and for which you have canned (or almost canned) solutions" (p. 855). Soloway's efforts offer a number of insights into programming instruction.

For several reasons, however, Soloway's work provided insufficient guidance for many of us. The list of canned solutions seemed incomplete. His advocacy of including "*explicit* instruction in 'vocabulary terms', such as mechanism, explanation, goal, plan, rules of programming discourse, plan composition methods, etc." [8, p.858] appeared to require adding content to an already loaded curriculum. Additionally, the suggestions offered for teaching the use of the canned solutions offered little pedagogical guidance.

Though Soloway's ideas seem less that directly applicable, they do provide significant insight into how programming is performed by experts. We now need better insight into how to turn novice programmers into expert programmers. The review of Soloway's work led to the suggestion that pattern-based instruction is a good way to provide initial instruction in programming. The remainder of the paper presents a case for patterning, and indicates our progress to date.

The Case for Patterns in Learning and Teaching

A major aspect of our work is the desire to have a model that will both explain and guide instructional practice. Developing curriculum is eased significantly if one has a model to suggest alternatives and to assist choosing from among alternatives. Spurred on by Soloway's ideas and the classic Chase and Simon [2] work in chunking of experience into remembered patterns, we proceeded to develop a pattern-based model for programming instruction.

We posit that learning can be viewed as the brain storing patterns associated with particular situations in particular domains of knowledge. Later performance, then, can be viewed as requiring a pattern-match between the present situation and previously learned ones. A problem solving activity will involve matching the current problem with past experience and retrieving an appropriate response or

solution (which may require some modification). This view fits well with our experience and with recent developments in the design of larger systems [5].

The model suggests that accomplished programmers will implicitly search their store of experiences or problem-patterns to produce a match for the current problem. The associated solution will then be more consciously examined to see if it can be modified to fit the current problem. If so, the modification begins. If not, the search resumes, perhaps more consciously. The solution patterns play an additional role. Experienced programmers, when designing programs, think using them rather than in smaller code segments or statements. The student or novice programmer, however, has no real chunks for thinking about programming. Rather, they will be thinking in terms of the statements they have learned, until their minds see patterns in the programs.

Using patterns to organize and focus instruction will encourage both instructors and students to de-emphasize syntax. Approaches to teaching natural language suggest this is feasible. Immersion is often recommended if one wishes to be able to communicate effectively in a second language. This emphasis on meaning and pragmatics seems to support teaching a programming language without concentrating on syntax. Students will see correct syntax in the examples presented and will adopt its use with little problem.

Student use of the patterns involves two skills only hinted at in the outline. First, students must learn to choose between similar patterns, e.g., process-all-items versus process-items-until-done. Additionally, students will need to develop skill at modifying patterns to fit specifics of the problem currently being worked on. Instruction will have to address such modifications. Students will need to learn that the pattern components have specific purposes and those purposes will indicate where modifications should occur. For example, there is only one place to look to change the way in a particular item will be processed in the process-all-items pattern.

Instruction should then provide students with: comprehensive exposure to problems; a set of solution patterns that can get associated with problems or problem types; and experience and expertise in adapting solution patterns to fit particular problems. We believe many students eventually develop patterns for their code and begin to think using them. Educationally, we would want to enhance the quality of the patterns developed and shorten the time required to develop them.

Deciding to address these issues does not supply answers to all the pedagogical questions. What patterns should be used? When, in the instruction, should they be introduced? Should patterns be taught before, as, or after students gain knowledge of syntax and semantics? How do the patterns interact with language features and what are the implications for instruction? Patterning can be used as the basis for an informal theory of learning— some of us often think about our teaching and everyday life in terms of patterning. The primary goal of this work, however, is to develop a relatively clearly-stated vision that can be used to test and guide instruction. We believe the patterning model provides that. We also believe it is consistent with current learning theory and brain research [4,6,9].

Work Toward Teaching Programming Via Patterning

Initial efforts several years ago expanded on the patterns suggested by Soloway's work. Over thirty code patterns that could be associated with problems, control structures, and data structures found in the first programming course were identified. One categorization of those patterns included: simple actions (e.g., prompted input, swapping values), conditionals (e.g., when, unless, dependent/nested conditionals, sequential conditionals), problems identified by loop purpose (e.g., counting, accumulating, file input, linear search, binary search), problems identified by loop control (e.g., sentinel vs counter control, multi-condition control), and more complex logical patterns (e.g., control break).

Progress using this large list was slow. Effort soon began to focus on a smaller set of patterns. Several guidelines relating to that smaller list emerged. First (in number, not necessarily importance), each pattern should involve or include a code pattern. Students must have access to a solution (a chunk of code) once they match the current problem with prior experience. Second, the patterns should be based on problems, not programming statements, control mechanisms, data structures, or algorithms. Students will encounter problems and seek code-patterns for them. The pattern-recognition that will occur in the mind will not be based on characteristics of the code, but rather characteristics of the problem. Thus, the pattern titles, actions, etc. should all be related to the problems involved. Third, the set of patterns should be comprehensive. They should cover all or nearly all the problems to be addressed in the course (or whatever context is used for their presentation). For initial learning, students should not be presented all the complexities of the eventual environment in which they will perform. Rather the stimuli they encounter should be restricted to those most essential for learning the patterns and their use. If the set of patterns is not comprehensive, these nonessential stimuli will confound the learning. Finally, the set of patterns should fill a particular niche in the hierarchy of abstraction. We want students to use the patterns as the building-blocks of or for programs. Individual statements are too low-level to provide useful chunks for thinking when writing programs. The suggested patterns fill that role nicely. They will not, however, suffice when building large systems. Thus, the set of patterns needs to be consistent in their level of abstraction and purpose. In this case, the proposed set of patterns provides the first level of building blocks for programs. They can be used for developing relatively small programs and modules and should allow us to provide students with a realistic design experience while introducing the lower-level language details.

Ultimately five patterns were settled upon. Individually, each is believed to both relate to problems and have a particular solution pattern. As a group, they are comprehensive and at the same level of abstraction. Figure 1 shows that set of patterns. Several introductory texts were examined to check for completeness of the set of patterns. The texts involved different languages—Pascal and COBOL. That search was rather informal in nature and involved seeking problems that could not be developed using the patterns. Certainly many problems are larger than the patterns but the patterns appear to provide a complete set of building blocks for all the programming problems examined.

Next, work began on instructional development. One part of that work involved seeking theory-based principles to be used in both instructional development and teaching. Thomas [10] described that activity. Suggestions include: focus on what students should be able to "do"; teach the big ideas; focus instruction on one idea at a time; teach strategies; incorporate quality review; encourage discovery learning; make assignments interesting, realistic, and motivating; and scaffold student learning. One aspect of that work was group discussion of the patterns and their use. It helped us to gain a fuller understanding of how the patterns might be used and taught. The discussion also helped us see how the principles can be used to guide pedagogical decision making. Some results of that effort are seen in the discussion above. Another result—an initial outline for the instruction—is presented in Figure 2.

Figure 1. The Patterns.

Process-One-Item
- get data to be processed
- process the data
- output the results

Guarded-Action
- if guard-condition is satisfied
  - take action

Alternative-Actions
- select appropriately from the following
  - condition 1, take action 1
  - . . .
  - condition n, take action n

Process-All-Items
- prepare for processing
- loop over (process-one-item):
  - get next item
  - process the item
  - prepare for next item
- perform closing action

Process-Items-Until-Done
- prepare for processing
- loop until done:
  - get next item
  - if appropriate
    - take action
  - prepare for next item
- if needed
  - process last or found item

Figure 2. Initial Course Outline.

- Introductions—course overview; computer operation concepts; programming via patterns
- Process-one-item pattern; program & statement syntax; program entry, editing, compilation, & execution; modifying patterns
- Guarded-action pattern; embedding patterns
- Alternative-actions pattern; implementation options; choosing between selection patterns
- Process-all-items pattern (with user- or redirected input); increased program complexity and embedded patterns
- Process-items-until-done pattern (using redirected input); choosing between iterative patterns
- Pattern review and consolidation
- Using the patterns with files
  - process-all-items (and embedded patterns)
  - process-items-until-done
- Pattern modification for arrays
  - process-one-item, guarded-action, alternative-actions
  - process-all-items (and embedded patterns)
  - process-items-until-done
- Pattern modification for pointers & linked structures (similar to above)
- Larger problems

Initial Experience

The course outline in Figure 2 was used in a Pascal course in the fall 1995 semester. The course was a service course, so it was more programming oriented than a majors course might be. A number of preliminary results have been identified.

It was much easier than expected for the instructor to emphasize patterning over syntax and semantics. In the past, discussion of problems, design, etc., generally deteriorated into discussion of language features. It is true, however, that this particular instructor had fully adopted patterning as a model for human learning. Other instructors might be less successful.

In some cases the patterning appears to have been treated as content. It is reasonable to name the patterns and to refer to them by name in instruction. In some cases during the instruction, however, it seemed that a particular pattern might be being treated as important in its own right and overshadowing the problems that used the pattern. Our current thinking is that this probably should not occur. The patterns should be explicit in instruction but not the focus of instruction.

Perhaps as a result of the tendency to treat patterns as content, too much time was spent on the initial patterns (process-one-item, guarded-action, alternative-actions). Later, too little time, or at least too little practice, was provided on the patterns. In particular, when introducing process-all-items, the pattern seemed so obvious and students seemed to follow the example that a relatively involved problem (one with embedded alternative actions) was assigned. With hindsight, clearly the students needed to practice several simple problems first.

On a related issue, it became apparent that students tended to start each new assignment from scratch rather than copying and modifying a prior solution. Students need to be encouraged to reuse their work. All problems should probably have a template for the solution or instructions for students to supply their own.

The most significant conclusion is that a different organization is needed. Recommendations for a different arrangement for the course material are shown in Figure 3. In particular, introducing user-input and then moving to redirected input (from a file) was thought to be conceptually simpler (or more accessible to students) than beginning with file input. Unfortunately, redirected input on the system used in this course added significant confusion to the process-all-items pattern as a read operation was required both before the loop and at the bottom of the loop. Students didn't see the generalization of the pattern and some may have missed the pattern altogether. The revision should allow for better development of patterning and also get students to more realistic problems more quickly.

Finally, course pedagogy should change. To the extent possible, the text should provide discussion and examples for learning syntax and semantics rather than static discussion attempting to illustrate the dynamic process of design. Of course, there would still need to be discussion of underlying concepts. However, if texts have a sufficient treatment of syntax and semantics, the instructor can deal with the patterns; adapting them to particular problems; and using them to illustrate underlying concepts.

Figure 3. Revised Course Outline.

- Introductions—course overview: computer operation concepts; programming via patterns
- Process-one-item pattern: program & statement syntax; program entry, editing, compilation, & execution
- Process-all-items pattern (with file input): EOF-controlled iteration (syntax & semantics); re-use process-one-item examples
- Guarded-action pattern: conditionals; embedding guarded actions in the process-all-items pattern
- Alternative-actions: implementation options; embedding alternative actions in the process-all-items pattern; choosing between the selection patterns

- Process-items-until-done pattern (probably using arrays after file input); choosing between iterative patterns
- Pattern modification for pointers & linked structures
- Larger problems
- Recursion?

Summary and Conclusions

This paper suggests that using a model in developing and delivering instruction is a good idea and presents one such model. Teaching based on some model is likely to be better than instruction that isn't. Furthermore, instructional research without such a model has little chance of producing truly useful results. The proposed model assumes learning and thinking involve a pattern-recognition process that compares experience with a current problem-solving situation. This pattern-based approach to programming instruction and its implications are described. Results from the preliminary trial of instruction based on the model is reported—they were mostly encouraging.

Using patterning provides an alternative focus for instruction and should allow for reduced class attention on syntax. Timing the introduction of the various language components must still be addressed and is not simple but the pattern model and other pedagogical principles can be of assistance. Our heuristic indicated that language features should be introduced as they are needed to support the patterns or as early as possible if they make assignments more realistic. For example, the *if* statement is introduced as part of the guarded-action and the *if-then-else* and *case* consstructs will occur with alternative actions The *record* structure could be introduced shortly after the process-all-items pattern is begun or much later in the course when extending patterns to include various language features. Files can be introduced almost immediately. Arrays, like records, could be introduced at various times. One could introduce modules early, if desired, as the process-all-items pattern could easily have a process-one-item pattern embedded in it. Organizing instruction by something other than statements requires getting used to but we see it as a continuation of Pattis' [7] call for organizing instruction in ways that make pedagogical sense.

During the instructional development, two possible approaches to teaching the patterns emerged. To date, we have been unable to decide which approach might be superior. One suggestion is to use specific examples of subpatterns to develop an understanding. For example, the process-all-items pattern could be taught using subpatterns such as counting items, accumulating an item sum, etc. and generalizing from them to the process-all-items pattern. In this case, the emphasis is on the patterns and modification is addressed implicitly but often. The alternate suggestion was to teach the subpatterns as examples of the general pattern, each requiring some modification. In this case, the emphasis is on modification of the patterns. Both approaches have merit and are supported by the model. This development is worth noting because it indicates that research in instruction should involve both theoretical and empirical aspects. We anticipate addressing this question further.

The instructional development relative to the patterns is still in its early stages. A Pascal course was taught using this approach in the fall of 1995. Both the basic idea of using patterns and the instructional principles were tested and suggestions made for improvement. Two other uses of the approach (in Pascal with a different instructor and in COBOL) are currently under way. The results of those efforts and the curriculum materials developed will eventually be available.

References

[1] Astrachan, O. and Reed, D. AAA and CS 1: The applied apprenticeship approach to CS 1. In *The Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, (March 2-4, Nashville, Tenn.) ACM/SIGCSE, New York, 1995, pp. 1-5.

[2] Chase, W.C. and Simon, H. Perception in chess. *Cognitive Psychology, 4* (1973), 55-81.

[3] Linn, M.C. and Clancy, M.J. The case for case studies of programming instruction. *Communications of the ACM, 35*, 3 (March, 1992), p.121-132

[4] Friedman, S.L., Klivington, K.A., and Peterson, R.W., Eds. *The Brain, Cognition, and Education*. Academic Press, 1986.

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[6] Hunt, E. The role of intelligence in Modern Society. *American Scientist, 83*, 4 (July-Aug. 1995), 356-369.

[7] Pattis, R.E. The "procedures early" approach in CS 1: A heresy. In *The Papers of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education*, (Feb. 18-19, Indianapolis, IN.) ACM/SIGCSE, New York, 1993, pp. 1-5.

[8] Soloway, E. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM, 29*, 9 (Sept. 1986), 850-858

[9] Sylwester, R. *A Celebration of Neurons: An Educator's Guide to the Human Brain*. Association for Supervision and Curriculum Development, Alexandria, VA, 1995.

[10] Thomas, S.R., Beck, W.E., Drake, J.M. East, J.P., Wallingford, V.E., A principles-based process for designing introductory programming courses. A poster presentation at the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education, (February 15-18, Philadelphia, PA).

Authors

J. Philip East is an Associate Professor of Computer Science Education in the Computer Science Department at the University of Northern Iowa. His professional interests include learning-based teaching, computing and society, and computer science education in general. His e-mail address is east@cs.uni.edu.

S. Rebecca Thomas is an Assistant Professor in the Computer Science Department at Marist College in Poughkeepie, NY. Her professional interests include artificial intelligence, particularly agent-oriented computing, and computer science education. Her e-mail address is Rebecca.Thomas@marist.edu.

Eugene Wallingford is an Assistant Professor of Computer Science in the Computer Science Department at the University of Northern Iowa. His professional interests include artificial intelligence, particularly knowledge-based systems, and computer science education. His e-mail address is w-allingf@cs.uni.edu.

Walter Beck is an Associate Professor of Computer Science in the Computer Science Department at the University of Northern Iowa. His professional interests include computing related mathematics, database systems, and computer science education. His e-mail address is beck@cs.uni.edu.

Janet Drake is an Assistant Professor of Computer Science in the Computer Science Department at the University of Northern Iowa. Her professional interests include software engineering, particularly requirements analysis, and computer science education. Her e-mail address is drake@cs.uni.edu.