Programming Pedagogy -- A Psychological Overview

Leon E. Winslow
University of Dayton, Dayton, Ohio
e-mail: winslow@saber.udayton.edu

Abstract

Can we turn novices into experts in a four year undergraduate program? If so, how? If not, what is the best we can do? While every teacher has his/her own opinion on these questions, psychological studies over the last twenty years have started to furnish scientific answers. Unfortunately, little of these results have been incorporated into curricula or textbooks. This report is a brief overview of some of the more important results concerning computer programming and how they can affect course design.

Introduction

Authorities from Plato onward have debated the subject of expertise and how it is acquired. It is only within the last thirty years or so that psychologists have studied experts to determine what characterizes an expert and what is necessary to become one. One conclusion is that many properties or characteristics of experts are independent of the field of expertise and, to a large extent, the way they became experts is also independent of the field. Many of these studies have been in the field of computer science with results directly applicable to the teaching computer science, but few people or even textbook authors seem to be familiar with this work. This paper gives an overview of some of the results and discusses some of the implications for teaching computing.

More precise definitions are given later, but for now, an expert is someone who is more than just proficient in a field and a novice is someone just entering an area. We are all experts in some area and novices in others. A field as broad as computing has many possible areas of expertise from general computing expertise to specific domain expertise. An authority in, say, networking may know little or nothing about compiler design. More important, from an educational point of view, a student may be an expert at CS1 material but a complete novice at upper division material such as network design or parallel programming. As will be seen, novices just entering a new area, regardless of level, have many traits in common and their needs are often very similar.

Most studies are limited to programming per se rather than computer science in general; perhaps because programming is what most practitioners do, perhaps because the other areas of computer science are so similar to other subject areas, such as mathematics or engineering, that the studies might be considered as redundant of ones in those areas. In other words, to study the pedagogy of, say, algorithm analysis or grammars one should look into studies on mathematics or engineering pedagogy, although many of the results presented here are domain independent.

While psychological studies of programming started in the 1970s, Sheil [Sheil, 1981] noted that many of the earlier studies of programming were methodologically weak because of the complexity of the programming process, the variability of individual programmers, claims beyond the data, and poor experiment design. Also many "studies of programming have found strong, persistent practice effects." Since then a number of research psychologists have entered the field and the methodology has improved considerably. [Gilmore, 1990a] is an example of several papers on experiment design and methodology to be used in programming studies.

More important, when study after study, regardless of the methodology, reaches the same conclusion, there must be some common, underlying phenomenon that is being measured. An example in point is the large number of studies concluding that novice programmers know the syntax and semantics of individual statements but they do not know how to combine these features into valid programs. Even when they know how to solve a problem by hand, they have trouble translating the hand solution into an equivalent computer program.

In other words, with the proper care, psychological studies are helpful in understanding students' difficul-

ties and the stages they must go through and can be helpful in designing a course or a complete curriculum.

This paper presents first some results comparing novice and expert programmers, then results on improving problem solving skills, followed by a section on pedagogy.

Novices vs. Experts

Various studies have concluded that novices:

- lack an adequate mental model of the area [Kessler and Anderson, 1989],
- are limited to a surface knowledge of subject,
- have fragile knowledge (something the student knows but fails to use when necessary) and neglect strategies [Perkins and Martin, 1986],
- use general problem solving strategies (i.e., copy a similar solution or work backwards from the goal to determine the solution) rather than strategies dependent on the particular problem,
- tend to approach programming through control structures, and
- use a line-by-line, bottom up approach to problem solution [Anderson, 1985].

Most studies differentiate between a task -- a goal with a known solution -- and a problem -- a goal with no familiar solution. What is a problem to a beginner may be a task to someone more advanced. Problem solving ability then is an important indicator of one's level of expertise and is used in the literature for comparison purposes between individuals and groups. It is an important characteristic of experts regardless of discipline.

Interestingly, studies in physics, mathematics, chess, and even basketball have shown that a number of these characteristics are common to novices regardless of the area. Notice also that many of these items are characteristic of novices regardless of level; for example, fragile knowledge and neglected strategies are common in graduate students working in new areas.

On the other hand, similar studies have found that experts :

- have many mental models and choose and mix them in an opportunistic way [Visser and Hoc, 1990],
- have a deep knowledge of their subject which is hierarchical and many layered with explicit maps between layers,
- apply everything they know,
- when given a task in a familiar area, work forward from the givens and develop subgoals in a hierarchical manner, but given an unfamiliar problem, fall back on general problem solving techniques,
- have a better way of recognizing problems that require a similar solution [Chi, *et al*, 1981; Davies, 1990] (presumably by applying templates or schemata),
- tend to approach a program through its data structures or objects [Petre and Winder, 1988],
- use algorithms rather than a specific syntax (they abstract from a particular language to the general concept),
- are faster and more accurate [Wiedenbeck, 1986; Allwood, 1986], and
- have better syntactical and semantical knowledge and better tactical and strategic skills [Bateson, Alexander & Murphy, 1987].

Again, many of these characteristics are independent of the field of expertise. Note that many of these characteristics require time and experience to develop, for example, deep knowledge or speed and accuracy.

Some characteristics are independent of the level of ability:

- Given a new, unfamiliar language, the syntax is not the problem, learning how to use and combine the statements to achieve the desired effect is difficult. (Anyone teaching, say, a functional language or an object-oriented language to an experienced programmer with no background in these areas is familiar with this phenomenon.)
- Learning the concepts and techniques of a new language requires writing programs in that language. Studying the syntax and semantics is not sufficient to understand and properly apply the new language.
- Problem solution by analogy is common at all levels; choosing the proper analogy may be difficult.
- At all levels, people progress to the next level by solving problems. The old saw that practice makes perfect has solid psychological basis.

One of the interesting conclusions is that it takes approximately ten years to turn a novice into an expert. So a four year undergraduate program is limited, at least for most students, to making some steps toward this goal; at best, it can prepare the student to make the rest of the progress on his/her own.

There is a continuum from novice to expert, but if we are to study the process and understand how one progresses along the continuum, it helps to break the continuum into pieces or stages. Different researchers use different stages, but the most commonly cited is probably the one by Dreyfus and Dreyfus [Dreyfus, 1985] who broke the continuum into five stages based upon capabilities in each stage:

Novice : Learns objective facts and features and rules for determining actions based upon these facts and features. (Everything they do is context free.)

Advanced Beginner: Starts to recognize and handle situations not covered by given facts, features and rules (context sensitive) without quite understanding what he/she is doing.

Competence: After considering the whole situation, consciously chooses an organized plan for achieving the goal.

Proficiency: No longer has to consciously reason through all the steps to determine a plan.

Expert: "An expert generally knows what to do based upon mature and practiced understanding."

Most of us would probably settle for a graduate who ranks between competent and proficient.

Rather than discuss the complete spectrum, Anderson [Anderson, 1983] presented three stages in learning a skill:

1. apply very general problem solving rules to the declarative information given in the task,
2. combine task-specific and general rules into a single rule and/or procedure, and
3. increase ability by practice.

In other words, we begin by combining the new information with general problem solving techniques and advance to generating new, task-specific problem solving procedures. While this may be new in psychology, mathematics pedagogy has used this process for generations.

[Patel and Groen, 1991] use a different three stage learning process for medical diagnosis:

1. develop a way to represent the necessary knowledge (models, memory, etc.),
2. learn to distinguish between relevant and irrelevant information, and
3. learn efficient use of information

Note that the problem solving skill being learned here is diagnosis and classification, useful skills in any science.

Some principles are common to all three:

1. Expertise starts with general problem solving skills and basic knowledge (facts, models, rules, etc.) of the field.
2. Expertise grows with practice which implies applying this knowledge to problem solving to learn:
   a. how to determine and use pertinent information, and
   b. to combine knowledge into more powerful problem solving capability.
3. The first two steps are repeated over and over again with new facts and kinds of problems added during each repetition.

These last principles suggest that novices need to learn the basic facts and general problem solving skills before they can really begin to master the field. A number of studies have determined that mathematics background is a good indicator of success in CS1. This might indicate that students who have mastered high school mathematics have also learned certain problem solving skills which other students lack and that putting all CS1 students on equal footing requires teaching general problem solving skills along with basic facts of syntax, etc.

Learning Problem Solving

Regardless of the student's level, practice is an important part of his/her education. Practice in language syntax or Finite State Machines is easy to furnish; programming problem solving practice is harder to design. Program problem solving can be broken into steps:

1. understand the problem,
2. determine how to solve the problem:
   a. in some form and
   b. in computer compatible form

   (note that novices have trouble going from a to b),
3. translate the solution into computer language program, and
4. test and debug the program.

Programming problems come from a wide range of problem domains and understanding the problem domain is critical [Adelson and Soloway, 1985]. An understanding of the problem to be solved and its solution space is critical for problem solving; bad models or limited solution spaces lead to bad solutions. The external problem domain models should include descriptions of the objects, their properties, allowed operations, and initial and final states. [Brooks, 1983], [Goldman, *et al*, 1977], [Miller and

Goldstein, 1977]. Pennington [1987] found that the best programmers constructed a mental representation of the real world problem domain. It must be noted that experts may know no more about a given programming language than advanced beginners, but they do know a lot more about the problem domain and problem solving.

These conclusions have different effects in different courses. In, say, operating systems the problem domain is equal to the course and furnishing good domain models, object classes, and problem solving techniques is a routine part of the course. In, say, systems analysis, the problem domains tend to be business or real time systems; neither domain is a direct subject of the course and many students may lack the necessary domain background so it is up to the instructor to extend the course to include this background. In other courses, some emphasis on general problem solving strategies are helpful. For example, we have found that many simple problems are easily expressed in terms of constructing a table; that is, the solution can be a table and constructing a sample table by hand gives insight into the steps required to solve the problem. Teaching this strategy helps students cope with step 2 above.

Every teacher has had troubles with students who do not read the problem carefully, or read it in a superficial manner. Mathematicians consider "word problems" one of the most difficult things to teach, yet programming is almost all word problems. The difficulty seems to be relating the words on the paper to something meaningful to the student. Practice helps, but the difficulty is bigger than computer science.

Once the problem solution is understood, the next step is translating the solution into computer terms. Many students can solve a given problem by hand but lack the ability to express the solution in computer terms. They need models which can be used to decompose the problem solution into steps they do know how to translate. A pattern that many have found useful is the

Process
Output

I prefer the ICO pattern version of this:

Input
Calculate
Output

because it emphasizes the three basic steps necessary for many simple problems. Once this is understood, the obvious extension is the IRT (looping) pattern:

Initialize
Repeat for each entry (in list, table, array, etc.)
Terminate

For example, the pattern

Initialize
Repeat for each row in table
    Process the row
end repeat
Terminate

suffices to solve many table problems. When these patterns are presented in progressively more sophisticated forms with many examples and student problems at each step, they learn to use these patterns as a model for decomposing a solution into manageable terms and have a strategy which is applicable for many undergraduate level, and real life, problems

These patterns are an example of the problem solving models that help students translate domain specific solutions into computer compatible solutions. Studies have shown that experts categorize problems by applying a large number of patterns or schemata which novices lack. Education which helps to develop schemata helps in solving real-world problems [Hesketh, Andrews and Chandler, 1989]. Multiple examples of the same schemata help [Gick and Holyoak, 1983]. While Brooks' "silver bullet" may be impossible, programming does need a systematic set of problem solving models each of which suffices for a large category of problems.

Study after study has shown that students have no trouble generating syntactically valid statements once they understand what is needed. The difficulty is knowing where and how to combine statements to generate the desired result. As noted earlier, experts think in terms of algorithms and not programs. The actual translation of an algorithm into a working program is a task, not a problem. Presumably the algorithms allow them to concentrate on the important features of the solution and ignore details which can be filled in later; in other words, it is a method for decomposing the problem solution into more manageable terms. Students should use algorithms for the same reason, yet many students seem to feel that they must be able to generate detailed programs without first developing an overview of the program.

The next step in programming is testing and debugging, a subject most texts skip or give only cursory coverage. Learning debugging:

- depends first and foremost on the student being able to read and understand programs (Note that this is different from writing programs -- a student may not be

able to understand a program he/she has written.); and
- requires a strategy; e.g., test one section at a time with good test data and plenty of print statements.

Gilmore [Gilmore, 1991b, Section 4] reviews the literature on novice *vs.* expert debugging. Studies have shown that there is very little correspondence between the ability to write a program and the ability to read one. Both need to be taught along with some basic test and debugging strategies.

A final step in problem solving is to consciously analyze what has been done along with how and why it has been done. All students seem to need continual reminding of this step.

Pedagogy

Linn and Dalbey suggest that there is an ideal chain for learning computer programming:

1. learn the syntax and semantics of one language feature at a time,
2. learn to combine this language feature with known design skills to develop programs to solve problems (this expands the students design skills and includes patterns and procedural skills such as planning, testing and reformulating), and
3. develop general problem solving skills.

This can be generalized to more advanced material:

1. learn one new feature or item at a time,
2. learn design skills combining old strategies and new ones, and
3. learn general problem solving skills plus those unique to this material.

Good pedagogy requires the instructor to keep initial facts, models, and rules simple and only expand and refine them as the student gains experience. One wonders, for example, about teaching sophisticated material to CS1 students when study after study has shown that they do not understand basic loops; more time spent on looping problems might pay a much larger return in the long run.

By the way, this trouble with loops indicates a deeper difficulty: almost any undergraduate can add a set of numbers or compute an average of a set of numbers; why can't over half of them write a loop to do the same operations? One wonders also what the results would be if similar experiments were performed with more advanced students. What do most operating systems students or database students lack after a course in the subject?

With the proper scaling of concepts, the same precept applies also to upper division students. The material should be introduced simply with much testing and analysis of results to determine exactly what the students are learning and what they are missing.

Models are crucial to building understanding. Models of control, data structures and data representation, program design, and problem domain are all important. If the instructor omits them, the students will make up their own models of dubious quality. Linn and Clancy [Linn and Clancy, 1992] recommend verbal descriptions, illustrations, and connections to other things as all being helpful.

The rules for combining operations into meaningful units are also important. If we don't teach basic paradigms of design (combining groups of commands into meaningful units), then students will make up their own (of doubtful quality) or attack each problem as if completely unknown. The author is continually amazed at the number of graduate students, many with degrees from leading undergraduate schools, who attack each problem as a new, unknown entity. They completely lack any understanding on the most basic level of design paradigms. Not surprisingly, these students take more time and produce poorer programs than students who know at least the basics. In the Dreyfus categorization, they are still advanced beginners with a large database of information but no systematic way to apply it.

Conclusions

Psychological studies of expertise in general and computer programming expertise in particular show that turning a novice into an expert is impossible in a four year program. Competence, however, is possible. Reaching this level requires mastering basic facts, features and rules and being able to consciously plan and carry through a problem solution in specified areas. The key to reaching this level is practice, practice, practice -- starting with simple facts and problems and working up to more and more complicated facts, strategies and problems. Adequate models of the problem space and the programming facts and strategies are essential to reaching this stage. Some successful problem solving models are included.

References

Adelson, B., and Soloway, E., The role of domain experience in software design, *IEEE Transactions on Software Engineering*, Vol. SE-11, No 11, November, 1985, pp. 1351-1360.

Allwood, C., Novices on the computer: a review of the literature, *International Journal of Man-Machine Studies*, Vol. 25, 1986, pp. 633-658.

Anderson, J., *The Architecture of Cognition*, Harvard University Press, 1983.

Anderson, J., *Cognitive Psychology and its Implications*, 2nd Ed., Freeman, 1985.

Bateson, A., Alexander, R., and Murphy, M., Cognitive processing differences between novices and expert computer programmers, *International Journal of Man-Machine Studies*, Vol. 26, 1987, pp. 649-660.

Brooks, R. (1983) Towards a theory of the comprehension of computer programming, *International Journal of Man-Machine Studies*, Vol. 18, pp. 543-554.

Chi, M., Feltovich, P., and Glaser, Categorization and representation of physics problems by experts and novices, *Cognitive Science*, Vol. 5, 1981, pp. 121-152.

Davies, S., The nature and development of programming plans, *International Journal of Man-Machine Studies*, Vol. 32, 1990, pp. 461-481.

Dreyfus, H. and Dreyfus, S., *Mind Over Machine*, Free Press, 1986.

Gilmore, D., (1990a) Methodological issues in the study of programming, in *Psychology of Programming*, Ed. by Hoc, J,, Green, T., Samurcay, R. and Gilmore, D., Academic Press, 1990, pp. 83-98.

Gilmore, D., (1990b) Expert programming knowledge: a strategic approach, in *Psychology of Programming*, Ed. by Hoc, J,, Green, T., Samurcay, R. and Gilmore, D., Academic Press, 1990, pp. 223-234.

Gick, M., and Holyoak, K. Schema induction and analogical transfer, *Cognitive Psychology*, Vol. 15, 1983, pp. 1-38.

Goldman, N. Blazer, R. and Wile, D. (1977) The use of a domain model in understanding informal process descriptions, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence.*

Hesketh, B., Andrews, S. and Chandler, P., Training for transferable skills: the role of examples and schema, *Educational and Training Technology International*, Vol. 26, 1989, pp. 156-165.

Hoc, J.-M., and Nguyen-Xaun, A., Language semantics, mental models, and analogy, in *Psychology of Programming*, Ed. by Hoc, J,, Green, T., Samurcay, R. and Gilmore, D., Academic Press, 1990, pp. 139-156.

Kessler, C. and Anderson, J., Learning flow of control: recursive and iterative procedures, in *Studying the Novice Programmer*, Ed. by Soloway, E. and Spohrer, J., Lawrence Erlbaum Associates, Publishers, 1989, pp. 229-260.

Linn, M., and Clancy, M., The case for case studies of programming problems, *Communications of the ACM*, Vol. 35, March, 1992, pp. 121-132.

Linn, M. and Dalbey, J., Cognitive consequences of programming instruction, in *Studying the Novice Programmer*, Ed. by Soloway, E. and Spohrer, J., Lawrence Erlbaum Associates, Publishers, 1989, pp. 57-82.

Miller, M. and Goldstein, I. (1977) Structured planning and debugging, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence.*

Patel and Groen in *Toward a General Theory of Expertise: Prospects and Limits*, Ed. by Ericson, K. and Smith, J., Cambridge University Press, 1991.

Pennington, N. (1987) Comprehension strategies in programming, in *Empirical Studies of Programmers: Second Workshop*, Ed. by Olson, Sheppard and Soloway, Ablex, 1987.

Perkins, D. and Martin, F., Fragile knowledge and neglected strategies in novice programmers, in *Empirical Studies of Programmers*, Ed. by Soloway and Iyengar, Ablex, 1986.

Petre, M. and Winder, R., Issues governing the suitability of programming languages for programming tasks. *People and Computers IV: Proceedings of HCI'88*, Cambridge University Press, 1988.

Sheil, B, The psychological study of programming, *Computing Surveys*, Vol. 13, No. 1, March 1981, pp. 101-120.

******Programming Continued On Page 25******

Collaboration between students was encouraged in the laboratory. We found, as have others [5, 9], that collaboration helps provide a student mentoring mechanism, improve performance due to peer pressure, and encourage students to ask questions only after trying to figure out a problem as a group. The students' attitudes were more supportive and less competitive.

4. Future Improvements

In the future, students should be better prepared for our CS-2 course. We are in the process of standardize the CS-1 course content, and we expect to see more transfer students with an object-oriented/C++ background.

A stronger coverage of the project management and team development strategies will be included to aid efficient operation of the team projects. This will necessitate deferring the topics of sorting and performance analysis to the Design and Analysis of Algorithms course.

We are examining a switch to a "cleaner" object-oriented language such as Eiffel. The advantages and disadvantages of this are still being debated within our department.

References

[1] Budd, Timothy A. (1994). *Classical Data Structures in C++*. Addison-Wesley Publishing Company.

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

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

[4] Headington, Mark and Riley, David. (1994). *Data Abstraction and Structures using C++*. D. C. Heath and Company.

[5] Parker, Brenda C. and McGregor, John D. (1995). "A Goal-oriented Approach to Laboratory Development and Implementation," *SIGCSE Bulletin*, 27(1):92-96.

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

[7] Reid, Richard J. (1995). "13th Edition of the list of languages used in the first course for Computer Science majors," informal survey available by ftp.cps.msu.edu: pub/arch/CS1_Language_List.Z.

[8] Wallingford, Eugene. (1996). "Toward a First Course Based on Object-Oriented Patterns," *SIGCSE Bulletin* 28(1):27-31.

[9] Yerion, Kathie A. and Rinehart, Jane A. (1995). "Guidelines for Collaborative Learning in Computer Science," *SIGCSE Bulletin* 27(4):29-34.

*********Programming From Page 22*********

Visser, W. and Hoc, J., Expert software design strategies, in *Psychology of Programming*, Ed. by Hoc, J., Green, T., Samurcay, R. and Gilmore, D., Academic Press, 1990, pp. 235-250.

Wiedenbeck, S., Processes in computer program comprehension, in *Empirical Studies of Programmers*, Ed. by Soloway, E. and Iyengar, S., Ablex, 1986, pp. 48-57.