In M. Petre & D. Budgen (Eds) Proc. Joint Conf. EASE & PPIG 2003 Pages 325-332

A Developmental Study of Cognitive Problems in Learning to Program.

Richard Tucker
Computing and Mathematics Sector
Banff and Buchan College
rtucker@banff-buchan.ac.uk

Abstract

Much of the research into the difficulties associated with learning to program has dealt with the differences in knowledge and behaviour between experts and novices; intermediate levels of competence have often been ignored. This paper describes an ongoing study into the problems faced by a group of students undertaking a year-long programming course. The study aims to investigate the cognitive difficulties which underlie those problems and to see how they change as the course progresses. The rationale behind the approach taken to the teaching is discussed, the key aspect being to make allowance for each student's individuality of understanding. The initial findings which are presented suggest that the problems faced by students can be easily misinterpreted. The students may not have learnt what the lecturer thought they were teaching, accounting for why so many students are still unable to program at the end of the course.

Background

Expert-novice differences

It is a truth universally acknowledged, that teaching computer programming is difficult. Much research has been carried out to investigate why this should be so, one important strand of which examines the knowledge and skills experienced programmers, or those termed ‘experts’, possess. This has provided data about expert behaviour in general and observations of such behaviour in many different domains ((McCormick (1999); Glaser (1999); bibliography in Hoffman (1998a,1998b); Feldman (1980)). Descriptions of those relating directly to programming skills may be found in Hoffman (1998b); Wiedenbeck et al. (1993); Riecken et al. (1991); Holt et al. (1987).

This type of research often contrasts the behaviour and skills of experts with those who are typically in the early stages of their careers, generally called ‘novices’. Valuable as this work has been, yielding much useful information about the relative performance of experts and novices, there has, however, been an overriding emphasis on the distinction between those two levels with scant attention paid to intermediate levels which must, by necessity, exist. This has been called the “binary expert-novice paradigm” in which little, if any, account is made of intermediate levels (Campbell, Brown and DiBello, 1992). Furthermore, the definition of the terms ‘novice’ and ‘expert’ are themselves subject to a wide variation in their interpretation: "one researcher's 'expert' will be another researcher's 'novice'" (ibid.). They must, therefore, be regarded as relative terms rather than having any absolute meaning in their own right.

What knowledge do student programmers need?

The content of many current programming curricula has been driven by this expert-novice distinction, emphasising that the knowledge and skills possessed by experts should dictate *what* should be taught. It assumes that novices are merely under-developed experts, and that

by being taught expert-like traits they will become experts themselves. One problem with this is that so much knowledge is needed to solve even the most trivial problem (Mayer, 1988). Students must learn to use all of this knowledge at the same time, making its acquisition all the more difficult as a result; in order to program they must make a "transition from simple computational attempts to the synthesis of the whole in a procedure or program", (Arzarello et al. (1993)). Adopting an approach that emphasises what needs to be taught, coupled with the binary novice-expert paradigm may expect too much from the students as they then need to become experts in order to begin.

How is knowledge acquired?

Glaser (1996) suggests the importance of another issue, saying that "In most studies of expertise to date, the acquired properties of expert performance have been described…only indirect attempts have been made to infer what properties of attained expertise might mean for the acquisition of competence." He continues, suggesting "a major principle or hypothesis underlying the acquisition of competence, which can be labelled a *change in agency*…for learning as expertise develops and performance improves".

The emphasis here is thus not so much on what experts know, but rather *how* they acquire that knowledge. As Campbell et al. (1992) say, "Looking for novice-expert differences in programming has not served researchers well. Maybe studying the development of expertise will serve better". There are clear implications here for the teaching of programming.

Implications for teaching programming

Glaser (ibid.) places an emphasis on learning and teaching processes and describes progression in terms of 3 interactive phases: a reliance on external support during the initial phases of learning; a transition period in which there is an "increasing of apprenticeship arrangements that offer guided practice and foster self-monitoring and learning of self regulation"; and the final phase of self-regulation in which, as a developing expert, the learner takes control of their own learning.

The use of approaches similar to this, which may be described more generally as cognitive apprenticeships, is discussed widely in the literature by, for example, Guzdial (1994); Arzarello et al. (1993); Marshall (1993). Linn and Clancy (1992) link the *what* and *how* aspects of teaching programming in a theme which runs through their work over a number of years, using case studies to show how a solution might be developed. In their earlier work they concentrate on the acquisition of design skills which, though crucial, are often neglected in the teaching of programming and note that students often "develop program design skills by inference and unguided discovery", stressing that "programming instruction tends to emphasize the product of the problem-solving process, but not the process itself".

This tendency raises an important point; one which has not been addressed by the inclusion in programming syllabuses of software engineering or software design units. Rather than being seen as an aid to designing a program, these units often serve only to place a further burden on the novice, introducing further 'products' for them to produce instead of helping the original 'process'. Furthermore, units such as these again concentrate on techniques used by experts with little concern for what may be more suitable for novices. Is it really any surprise that students should find it difficult to understand a formal design methodology which has been designed to aid experts rather than address novices' difficulties?

Linn and Clancy then synthesise a number of principles involved in programming, based on how experts approach programming, that are often "only tacit in programming courses" (Bell, Linn and Clancy (1994)). They attempt to teach these interactively, "encouraging students to autonomously make sense of programming" and stress the importance of an apprenticeship approach to teaching programming.

There are two important aspects of learning to program that make apprenticeship approaches particularly suitable, beyond their inherent emphasis on teaching problem-solving skills. First, programming is an inherently abstract skill. The programmer must understand abstract concepts that are difficult to comprehend. In order to understand those abstractions, therefore, the novice must possess some expert-like traits, most specifically conceptualisation, yet to acquire them they must gain some experience of programming. Formal design methodologies do not help here, and may exacerbate the problem by introducing further abstractions to confuse the student.

The second aspect is that in developing an understanding of programming and its abstractions, each student will bring to bear their own prior experience which will be unique to themselves. This constructivist view of learning is discussed in more detail by Campbell et al. (1992). Thus, each student will develop their own, perhaps unique, approach to the abstraction. This *individuality of understanding* in the programming context is noted by Taylor (1987), cited in Colley and Beech (1989), who states that most approaches to studying programming ignore students' backgrounds, i.e. they treat everyone in the same way and concentrate on turning them into putative standardised experts. Van der Veer (1992) investigates the wide variety of mental models that students use, yet programming courses and text books tend to suggest, if only by not considering alternatives, that there is a single, notionally ideal, solution to any problem. At best, such approaches do not foster the development of another key expert trait, that of considering multiple representations of problems (see, for example: Hoffman (1998a)). (Bell, Linn and Clancy's case studies approach does encourage considerations of alternatives through asking 'stop and think / help / predict / consider' questions (1994), although even then there may be a tendency for students to regard as *the* solution any with which they are presented.) At worst, the student gets confused in trying to understand a view of the abstraction to which they do not personally relate.

A lack of attention to this individuality of understanding is also present in the approach taken to program design. In their investigation, Holt et al. (1987) concluded that there were "strong individual differences in the content and structure of the mental models of programmers" but again no allowance for these differences are made when teaching formal program design methodologies.

Some of the problems here have been studied in work associated with the value of analogies. For example DuBoulay (1989) discusses some difficulties with analogies that are commonly used when teaching programming. Allwood (1986) cites Halasz and Moran (1982) who argue against the direct use of analogies since the computer system falls outside the intended analogy and it is difficult for the user to know which aspects are to be included and which are not. If students adopt, in part or as a whole, an inappropriate analogy then they will have to spend time later on unravelling their misunderstanding and modifying their internal knowledge structures.

If, then, students are presented with a method of instruction that does not take account of their own peculiar understanding of the problem being considered, what might they learn as a result? In the context of programming the answer may be 'not much' or 'not as much as you think'. Marshall (1993) provides a relevant comment here when, focussing on the nature of knowledge gained during introductory instruction about arithmetic word problems, she says that it is a "rare instance in which all learners learn exactly the same thing from a single instructional lesson". Her investigation again emphasises the power of abstract understanding and stresses the importance of the very first example of a concept since it provides the scaffolding for the understanding that follows later on. If there is an individuality of understanding, then it is crucial that allowance is made for it in the early stages of instruction and the student who gains an acceptable understanding of the abstraction first will be the student who makes the fastest and most effective progress.

In conclusion, two fundamental misconceptions underlie the way in which teaching programming is often approached: that novices should be taught directly what experts know and that no allowance is made for each student's individuality. What should be taught is what is teachable given each student's level within the developmental process of understanding the abstract concepts that are involved.

Current Research

Two questions that then arise are, "What problems do students face when learning to program?" and "How do the nature of those problems develop over time?". My current research aims to provide some answers to those questions by investigating the underlying problems that are encountered by a group of students undertaking a year-long course of instruction.

Twenty-one students are involved in the study, taking software design and programming units as part of an HNC in Computing at a college of Further Education. Some of the students have previously completed an introductory programming course; some are complete beginners. The language being taught is Visual Basic.

The study comprises two stages. The first of these involves taping conversations with students when they have a problem and need help in solving it. During these interactions I am seeking to understand the underlying reasons for the student experiencing difficulties. I am also collecting samples of the students' solutions to formative exercises in order to check on the students' progress and monitor the development of their programming skills. On the basis of those solutions, further conversations are then recorded as necessary. The recordings will be transcribed and analysed later to seek commonalities across the nature of the problems and gauge how they changed during the duration of the course. I am currently about halfway through this stage.

In the second stage I intend to see if the results from the first can be generalised by examining the difficulties experienced by students undertaking the same course at other Further Education colleges using a variety of programming languages and teaching approaches. Anecdotal evidence would seem to suggest students everywhere encounter the same types of difficulties.

Approach taken to delivering the course

Over the past 3 years I have been developing an approach to teaching programming based on the background work described earlier. The course is centred on the development of a case-study program. This follows Linn and Clancy's (1992) rationale, but it differs in that I never present any form of idealised solution to the case study. Indeed, I never present any kind of solution but merely discuss the student's own as they develop. Individual differences in approach are thus addressed in this way; a reversal of roles in the master-apprenticeship relationship with the student asking the questions and the master providing guidance as appropriate.

There needs to be some overall structure to the introduction of programming techniques and constructs, but I try to keep that structure as loose and flexible as possible. Topics are typically introduced with a brief lecture, lasting between 5 and 15 minutes. The students are then presented with a number of problems to solve which involve the application of the new topic but in contexts which aim to show why the technique is useful rather than simply providing practice in how it is used. Very few handouts are provided, students being encouraged to write their own notes using whatever language makes sense to them. Some of the problems are linked back to previous lessons to encourage code reuse and transfer (Hoadley et al. (1996)) whereas others require the students to extend their knowledge beyond what has been taught. Whenever a student encounters a difficulty, this is dealt with by a

progressive system of ‘prompts, hints and provides’ similar to that described by Perkins and Martin (1986).

This latter type of exercise addresses the individuality of each student’s approach directly as the new knowledge required for the solution is only introduced as the student sees its need and application. (Guidance away from solutions which are too complicated or judged to be unfeasible may be necessary, but so far that has been a rare occurrence.) The solutions to problems are thus inherently student-orientated: programming techniques and concepts are introduced in a context which is of the student’s own making and therefore more likely to be understood. Hiebert et al. (1999) describe the benefit of ‘problematizing’ teaching mathematics in this way.

The rate at which topics are introduced is not predetermined either; students are not rushed through the material. This is particularly important during the early stages as they struggle to develop an initial understanding of the concepts which are involved in programming.

Initial findings

I have yet to start a formal analysis of the data, but the following observations may be made.

The symptom of a problem is rarely its cause.

For example, students do not, generally, omit quotation marks around a string constant because they have merely forgotten to do so. More typically, they may do it because they have not understood the need for that syntactic device to distinguish the constant from a variable reference. A student’s failure to resolve this problem for themselves can indicate a fundamental lack of understanding of variables.

Further evidence of this may be found in studies such as Spohrer and Soloway (1986) investigating the frequency of bugs in students' programs in which they conclude that certain types of bugs occur more frequently than others. Has the ability of student programmers increased dramatically as a result of identifying those common bugs?

Students can quite easily deceive themselves, and the lecturer, by writing code which they do not understand.

This often occurs through the student applying trial and error to get a program to work, but some cases here have involved quite complex code (with multiple nested control structures for example) which the student has developed by themself. The miscomprehension only becomes apparent later when the student attempts to reuse the code, or parts of it, elsewhere when the need for a change such as altering the condition in a selection statement can be an insurmountable problem for the student. In cases like this the misunderstanding means the code cannot be reused elsewhere. As Hoadley et al. (1996) say, such reuse is closely linked to the code comprehension abilities of the student.

More generally, the implication here is that just because the students have completed all the exercises, it does not mean that they have understood what they have done.

What students learn can be very different from what the lecturer thought they were teaching.

For example, when teaching the concept of ‘events’ in Visual Basic some students associated the concept with the control (such as a CommandButton) to which the event applies rather than the event itself. This leads to errors later when they regard their code as ‘control-driven’ rather than ‘event-driven’.

The two preceding observations relate to a special kind of what Perkins and Martin (1986) term ‘inert knowledge’, “knowledge that a person has, but fails to muster when needed". The problem here is that that knowledge may be incorrect or misunderstood, in which case they

cannot transfer it successfully to a new situation. Indeed, they may not be even able to see the possibility of a transfer since they have not understood what they had programmed in the first instance, making retrieval of that knowledge unlikely. In many cases the phrase 'misunderstood knowledge' is more appropriate than 'inert'. Note, however, that in Perkins and Martin's study this type of problem accounts for nearly half of those identified. A similar interpretation can be taken of their 'misplaced knowledge': it is misplaced because it wasn't understood in the first place. The transfer of knowledge cannot therefore take place because there is no knowledge to transfer. Misunderstandings also seem to be extremely persistent and resistant to change. Students can continue to apply an inappropriate understanding of some concept despite numerous attempts to dissuade them otherwise. This observation emphasises some of the problems associated with analogies that were mentioned earlier.

I suggest that this problem is the principal reason why students are unable to write end-of-term projects despite completing all the coursework during the year. The lecturer should not overestimate how much the student has actually understood. Some misunderstandings can lie dormant for months. For example, the problem with quotation marks mentioned in the first observation only surfaced for one student during week 10 of the course.

Don't rush: slow starters can catch up quickly once they begin to understand the abstractions.

One student struggled to make any progress for 7 weeks. Between weeks 9 and 10, however, they caught up with the rest of the class. I am trying to investigate what caused this sudden increase in the rate of progress but my data-gathering methods are not sufficiently retrospective to be of much use. The student's perseverance in reading books and copying code examples seems, however, to be significant.

'Provides' (giving students a solution to their problem) rarely work.

If this is done the student's understanding is not increased. Rather, they are presented with a piece of code which they may not understand and therefore cannot integrate into their developing knowledge base.

'Fragile knowledge' is a good thing! (For the learner)

The principles behind causing 'cognitive conflict' in the student's mind is described more fully in Adey and Shayer (1994). The ever-present 'fragile knowledge' described by Perkins and Martin (1986) provides a fertile ground for this approach. Indeed the question could be asked, "Do students ever develop knowledge that is not fragile in some way?" If the problem of fragile knowledge is not going to go away it should be turned to the student's advantage. Rather than trying to reduce it as Perkins and Martin suggest, I believe its existence should be accepted as a vital part of the learning process.

Summary

When a student encounters a problem while learning to program, the reason why they cannot resolve the problem can be very different from that suggested by its symptoms. The underlying cause can also be very different from that expected by the lecturer and may well be individual to each student, especially when it is due to an instance of misunderstood knowledge, the persistence of which can cause ongoing difficulties.

As this work continues to the end of the year-long course I will be able to investigate further the development of understanding, or perhaps more appropriately of misunderstanding, of the students. This will provide further insights into why so many students are still unable to program at the end of their course. The approach taken to the teaching certainly matters (a strict syntax-orientated bottom-up approach simply does not work for many students) but it may also be that the curriculum demands too much from the students when they are at the lower levels of their development as programmers.

Bibliography

Adey, P. and Shayer, M. (1994). *Really raising standards: cognitive intervention and academic achievement.* London: Routledge.

Allwood, C. M. (1986). Novices on the computer: a review of the literature. *International Journal of Man-machine-Studies*. (25) 633-658.

Arzarello, F. et al. (1993). Learning to program as a cognitive apprenticeship through conflict. In E. Lemut, B. du Boulay and G. Dettori (eds.), *Cognitive models and intelligent environments for learning programming*. Berlin: Springer-Verlag.

Bell, J.E., Linn, M. C. and Clancy, M. (1994). Knowledge integration in introductory programming: CodeProbe and interactive case studies. *Interactive Learning Environments*. 4(1) 75-95.

Campbell, R. L., Brown, N. R., and DiBello, L. A. (1992). The programmer's burden: developing expertise in programming. In R. Hoffman (ed.), *The psychology of expertise.* New York: Springer-Verlag.

Colley, A. M. and Beech, J. R. (eds.) (1989). *Acquisition and performance of cognitive skills*. Chichester: Wiley.

DuBoulay, B. (1989). Some difficulties of learning to program. In E. Soloway and J. C. Sphorer (eds), *Studying the novice programmer*. 283-300. New Jersey: Lawrence Erlbaum.

Feldman, D. H. (1980). *Beyond universals in cognitive development.* Norwood, NJ: Ablex.

Glaser, R. (1999). Expert knowledge and processes of thinking. In R. McCormick and C. Paechter (eds.), *Learning and knowledge*. 88-102. London: Sage.

Glaser, R. (1996). Changing the agency for learning: acquiring expert performance. In K. A. Ericsson (ed.), *The road to excellence – the acquisition of expert performance in the Arts and Sciences, Sports and Games.* 303-311. New Jersey: Lawrence Erlbaum.

Guzdial, M. (1994). Software-realized scaffolding to facilitate programming for science learning. *Interactive Learning Environments*. 4(1) 237-264. New Jersey: Ablex.

Halasz, F. and Moran, T. P. (1982). Analogy considered harmful. *Proceedings of human factors in computer systems*. Gaithersburg, MD: ACM.

Hiebert, J., Carpenter, T. Fennema, E. et al. (1999). Problem solving as a basis for reform in curriculum and instruction: the case of mathematics. In P. Murphy (ed.), *Learners, learning and assessment.* 151-170. London: Paul Chapman.

Hoadley, M., Linn, M. C., Mann, L. M. and Clancy, M. J. (1996). When, why and how do novice programmers reuse code? *Empirical studies of programmers, sixth workshop.* New Jersey: Ablex.

Hoffman, R. R. (1998a). How can expertise be defined? Implications of research from cognitive psychology. In R. Williams, W. Faulkner and J. Fleck (eds.), *Exploring expertise: issues and perspectives*. 81-100. Basingstoke: Macmillan.

Hoffman, R. R. (1998b). *The psychology of expertise: cognitive research and empitical AI.* Berlin: Springer-Verlag.

Holt, R. W., Boehm-Davis, D. A. and Chultz, A. C. (1987). Mental representations of programs for student and professional programmers. In G. M. Olson, S. Sheppard and E. Soloway (eds.), *Empirical studies of programmers: second workshop*. 33-46. NJ: Ablex.

Linn, M. C. and Clancy, M. J. (1992). The case for case studies of programming problems. *Communications of the ACM.* 35(3).

Marshall, S. P (1993). Statistical and cognitive models of learning through instruction. In S. Chapman and A. L. Meyrowitz (eds.), *Foundations of knowledge acquisition. Cognitive models of complex learning.* 119-146. Boston: Kluwer.

Mayer, R. E. (1988). From novice to expert. In M. Helander (ed.), *Handbook of human-computer interaction*. 569-580. Amsterdam: Eslevier.

McCormick, R. (1999). Practical knowledge: a view from the snooker table. In R. McCormick and C. Paechter (eds.), *Learning and knowledge*. 112-135. London: Sage.

Perkins, D. N. and Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In E. Soloway and S. Iyengar (eds.), *Empirical studies of programmers*. 213-229. New Jersey: Ablex.

Riecken, R. D., Koenemann-Belliveau, J. and Robertson, S, P. (1991). What do expert programmers communicate by means of descriptive commenting? In J. Koenemann-Belliveau, T. G. Moher and S. P. Robertson (eds.), *Empirical studies of programmers: fourth workshop*. 177-195. NJ: Ablex.
Spohrer, G. and Soloway, E. (1986). Analysing the high frequency bugs in novice programs. In E. Soloway and S. Iyengar (eds.), *Empirical studies of programmers*. New Jersey: Ablex.
Taylor, J. (1987). A study of novices programming in PROLOG. Unpublished PhD thesis, Sussex University.
Van der Veer, G. C. (1992). Mental representations of computer languages – a lesson from practice. In Lemut et al. (eds.), *Cognitive models and intelligent environments for learning programming*. 20-33. Berlin: Springer-Verlag.
Wiedenbeck, S., Fix, V. and Scholtz, J. (1993). Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*. 39(5) 793-812. Academic Press.