Computers in Human Behavior, Vol. 6, pp. 273-289, 1990
Printed in the U.S.A. All rights reserved.

0747-5632/90 $3.00 + .00
Copyright © 1990 Pergamon Press plc

Automation and Schema Acquisition in Learning Elementary Computer Programming: Implications for the Design of Practice

Jeroen J. G. Van Merriënboer and Fred G. W. C. Paas

University of Twente

Abstract — *Two complementary processes may be distinguished in learning a complex cognitive skill such as computer programming. First, automation offers task-specific procedures that may directly control programming behavior, second, schema acquisition offers cognitive structures that provide analogies in new problem situations. The goal of this paper is to explore what the nature of these processes can teach us for a more effective design of practice. The authors argue that conventional training strategies in elementary programming provide little guidance to the learner and offer little opportunities for mindful abstraction, which results in suboptimal automation and schema acquisition. Practice is considered to be most beneficial to learning outcomes and transfer under strict conditions, in particular, a heavy emphasis on the use of worked examples during practice and the assignment of programming tasks that demand mindful abstraction from these examples.*

INTRODUCTION

Computer programming at an elementary level is rapidly becoming a part of the high school curriculum. However, there is much evidence for low learning outcomes after relatively short elementary programming courses of 10-50 lessons (Linn, 1985; Pea & Kurland, 1984). After these courses, most students still have an incomplete or incorrect mental model of the working of a computer (DuBoulay, 1986; Pea, 1986), a fragile knowledge base related to the basic commands and

Requests for reprints should be addressed to Jeroen J.G. van Merriënboer, University of Twente, Department of Instructional Technology, P.O. Box 217, 7500 AE Enschede, The Netherlands.

syntax of the programming language (Perkins & Martin, 1986; Putnam, Sleeman, Baxter & Kuspa, 1986; Sleeman, Putnam, Baxter & Kuspa, 1986), a serious lack of programming language templates or programming plans (Dalbey & Linn, 1985), and ill-developed procedural skills, such as for planning the solution or testing and debugging the program (Kurland, Pea, Clement & Mawby, 1986).

Thus, students do not learn to program very well in these elementary programming courses and most teachers are reasonably satisfied if the students at least acquire a vague notion of what programming is like and what it may be used for. Given these low learning outcomes, it is not surprising that most research on far transfer effects of elementary computer programming to other domains, or, more generally, an impact on higher level cognitive skills, yielded negative results (e.g., see Goodyear, 1987).

The goal of this article is to present instructional design principles that may augment learning outcomes and increase the possibility that transfer effects at least occur within the programming domain. This goal is subject to some restrictions. First, the design principles are limited to elementary computer programming in schools, where the target group mainly consists of prenovices and the available instructional time is severely limited. Second, and more important, the design principles are limited to the *design of practice*. The great majority of research on teaching elementary programming pertains to the presentation of particular information that is considered to be relevant to performance of the programming skill, such as the explicit teaching of concrete computer models (Mayer, 1981, 1982), programming language templates (Soloway, 1985), and debugging models (McCoy Carver & Klahr, 1986). Whereas such research certainly is valuable, an important claim of this article is that more attention should be paid to the conditions under which the programming skills are actually practiced.

The structure of this discourse is as follows. In section two, *automation* and *schema acquisition* are described as two prevalent processes in learning elementary computer programming and their demands to effective practice as well as their effects on learning outcomes and transfer are discussed. In section three, the major shortcomings of current programming instruction are identified and instructional design principles are presented that do meet the requirements to practice of both automation and schema acquisition and thus may improve learning outcomes and increase the possibility that transfer effects occur within the programming domain. In particular, this paper will focus a heavy emphasis on the use of worked examples during practice and the assignment of programming tasks that demand mindful abstraction from these examples. Finally, section four contains a discussion of the proposed approach and its inclusion in programming curricula, and a reflection on its main implications for teaching programming.

LEARNING COMPUTER PROGRAMMING

Learning computer programming means both learning procedures to accomplish various goals and learning the information that is relevant to these procedures. As a first observation, expert programmers can perform many procedures without noticeable effort because they are able to respond in a highly reflexive manner to abstract features of problems. However, their skill clearly is more than the sum of its automatic parts; when experts are confronted with new programming problems for which they have no automatic procedures available, they can rely on an enormous amount of programming knowledge that may be used by more general problem solving methods to reach a solution. Thus, besides the development of automatic procedures, the acquisition of highly structured knowledge, or schemata, plays a significant role in learning a skill like computer programming. In the next sections, the processes of automation and schema acquisition, their implications for the design of practice, and their effects on learning outcomes and transfer will be further elaborated.

Automation

Automation leads to highly task-specific procedures that may directly control programming behavior. In current cognitive research, such procedures are usually referred to as productions or condition-action pairs. The conditions specify various problem specifications or particular programming goals; the actions can be to embellish the problem specification, to set new subgoals, or to write or change programming code. As a result of the availability of task-specific procedures, experts can almost automatically reformulate and decompose familiar problems in subproblems that have known solutions, and they can effortlessly generate programming code to reach low-level goals, such as printing values, doing loops, or making decisions (Anderson, Farrell, & Sauers, 1984).

The development of automatic processing. The development of task-specific procedures is a lengthy process, that may be seen as a transition from controlled to automatic processing (Shiffrin and Schneider 1977; Schneider & Shiffrin, 1977). In the early stage of learning a complex cognitive skill, the learner usually receives information about the skill that may be used by general procedures, or "weak" problem solving methods, to generate behavior. The generality of those procedures refers to the fact that they make no reference to any particular knowledge domain; instead, they are able to interpret a wide range of newly acquired information to generate behavior. Such controlled processing has the advantage of flexibility because a learner can be circumspect about the behavioral implications of using the newly acquired knowledge. However, performance is low because controlled processing has the disadvantage that it works slowly and it may lead to serious errors due to processing overload.

Anderson (1983, 1987) identified *knowledge compilation* as an important process to make the transition from controlled to automatic processing possible. With practice, knowledge compilation creates procedures that eventually may directly control programming behavior. Knowledge compilation both includes the incorporation of newly acquired knowledge in new task-specific procedures and the "chunking" of procedures that consistently follow each other in solving particular problems. Knowledge compilation produces a considerable speedup in performance and implies a reduction of processing load because newly acquired knowledge need no longer be retrieved from memory and held active to be interpreted by more general procedures.

Complete automatic processing may be reached as learning proceeds further through a tuning process that strengthens the task-specific procedures with every successful application, so that situation-driven procedures become available that directly control programming behavior. Automatic processing works fast, with minimal errors, and with low demands on processing capacity so that cognitive resources become available for other aspects of the task. However, automatic processing may be disadvantageous as well because there is the ever present danger of action slips to occur if particular stimulus input triggers nonintended procedures (Norman, 1981).

Automation and the design of practice. As automation is the result of practice, skills can only be acquired by doing them. This principle of "learning by doing" has some important implications for the design of practice. First, an expert programmer is believed to have available tens of thousands of highly task-specific procedures (e.g., Brooks, 1977). The development of such a broad range of highly task-specific procedures, which underlies flexibility in programming behavior on a high performance level, requires lengthy training as well as a high variation in training.

Second, the detailed procedural knowledge is likely to be highly implicit and not easily verbalized, so that teachers may have difficulty explicating such knowledge. An effective alternative way to communicate the knowledge and to shorten the training for automation of the skill is the use of worked examples. In this respect, Anderson *et al.* (1984) and Pirolli and Anderson (1985) reported that students made a highly selective use of instructional materials during practice. In particular, they used concrete examples of problem solutions that were similar to the solution of the problem at hand and that had the form of concrete computer programs. Students used these worked examples as a kind of *concrete* schemata to map their new solutions. The key to this use of worked examples is interpreting the example by general procedures and mapping it onto the current knowledge of programming to create new solutions. Such interpretation of worked examples is a powerful tool in guiding programming behavior. But most importantly, the information that comes from the worked examples may be incorporated, or compiled, into new task-specific procedures. Thus, the use of worked examples initially bridges the gap between current knowledge and programming behavior and facilitates the development of task-specific procedures and, eventually, automation.

Automation and transfer. With regard to the effects of automation on transfer, a distinction must be made between near and far transfer. Mayer and Greeno (1972) introduced this distinction to indicate the extent of similarity between the new setting and the original training setting. For the purposes of this paper, near transfer is defined as transfer of programming skills within the programming domain, such as the ability to solve new programming problems; far transfer is defined as the transfer of programming skills outside the programming domain, such as the ability to apply learned top-down design techniques in writing an essay. This distinction is closely related to the issue of context-dependent versus context-independent strategies in programming (Perkins & Salomon, 1989), because far transfer assumes an excessive decontextualization of acquired skills. The present article is limited to near transfer, that is, to the transfer of skills within the context of programming.

Automation may explain such transfer by the overlap of task-specific procedures that were learned in the original task but that are also applicable in performing the transfer task. In fact, this explanation is closely related to the associationist theory of *identical elements* (Thorndike & Woodworth, 1901), which claimed that transfer from one task to another would only occur when both tasks shared identical elements. Whereas it never became clear what exactly was meant by identical elements, it was usually interpreted to mean something like stimulus-response pairs. In current cognitive research, the identical elements are usually interpreted in terms of productions (e.g., Singley & Anderson, 1985, 1988). The availability of automatic procedures predicts transfer in so far as the procedures that are learned in the training task are identical to the procedures that are needed for performing the transfer task. Salomon and Perkins (1987) refer to this transfer mechanism, which results from extensive practice and automation, as *low-road* transfer. It is limited by the triggering stimuli that will activate automated performance and hence requires varied practice to reach transfer.

In increasingly further transfer within the programming domain, there is a decreasing overlap of task-specific procedures between the original task and the transfer task. Thus, automation cannot directly explain the ability to solve new, not previously encountered—aspects of—programming problems. However, automatic processing of certain aspects of the task makes very low demands on processing capacity, so that cognitive resources become available for other controlled processes that *may* lead to such transfer. This side-effect of automation can be argued to be particularly important for a problem-solving intensive task such as computer programming. The interpretation of schemata that provide analogies for solving new problem situations is a good example of a form of controlled processing that may occur due to automation, and lead to further transfer within the programming domain. This process will be discussed in the next section.

Schema acquisition

Schemata can be conceptualized as cognitive structures that allow particular objects, events, or activities to be assigned to general categories. Thus, schemata provide general knowledge that can be applied to particular cases. Due to the availability of schemata, expert programmers are not only able to fluently perform familiar programming tasks by the use of highly task-specific procedures, but also to interpret unfamiliar situations in terms of their generalized knowledge. For instance, they may rely upon a good notion of the working of the computer to make their programs more efficient, their clear view of the design process in program development to guide their programming behavior, and their extensive knowledge base of programming plans to improve their problem decomposition and program composition.

The acquisition of several kinds of schemata is also relevant to learning elementary computer programming (Rist, 1989). For instance, a general design schema should be developed to provide abstract knowledge concerning the processes involved in generating a good design and its overall structure (e.g., Jeffries, Turner, Polson & Atwood, 1981). The design schema may then be used recursively to generate a decomposition of the problem into more and more detailed modules in a process of "stepwise refinement", which leads to a top-down, breadth first expansion of the solution. The design process continues until programming code has been identified for each of the subproblems.

Programming plans are generally considered to be a particularly important kind of schemata to acquire in elementary computer programming (Ehrlich & Soloway, 1984; Soloway, 1985). These programming plans are learned programming language templates, or stereotyped sequences of computer instructions, that form a hierarchy of generalized knowledge. High-level programming language templates (such as a general input-process-output plan) may be applied to a very wide range of programming problems, whereas medium level templates (such as a looping structure with an initialization above the loop) and low-level templates (such as a statement to print the value of a variable) are applicable to increasingly smaller ranges of (sub)problems. Thus, programming plans provide, within the programming domain, a hierarchy of increasingly context dependent strategies that may guide a process of "templating" in the creation of solutions to posed problems. In our discussion of schema acquisition, we mainly focus on the learning of such programming language templates during practice.

The development of schemata. In the prenovice stage of learning programming, the learner has neither task-specific procedures nor useful cognitive schemata available. Thus, the learner has to apply very general, weak problem solving methods to perform the programming task. As discussed in the previous section, a result of practice is that task-specific procedures are compiled that will significantly increase performance on subsequent problems. But in addition, and often simultaneously, schemata may be acquired that offer analogies, or abstract categories of problems and solutions, that may guide subsequent problem solving behavior.

Learning processes may either create new schemata or adjust existing schemata to make them more in tune with experience. For example, inductive processes can be described (e.g., Carbonell, 1984, 1986) that either extend or restrict the range of applicability of schemata. A more generalized schema may be produced if a set of successful solutions is available for a class of related problems, so that a schema may be created that abstracts away from the details; a more specific schema may be produced if a set of failed solutions is available for a class of related problems, so that particular conditions may be added to the schema which restrict its range of use. Recent research points out that such schema acquisition is a form of controlled processing, that is, it is subject to strategic control (e.g., Anderson, 1987; Proctor & Reeve, 1988). Consequently, compared to automation, which slowly develops and is mainly a function of the amount of practice, the acquisition of schemata such as programming plans may rapidly occur but requires the investment of effort, or, conscious attention and mindful abstraction from the learner.

After useful schemata have been developed, they may be used as analogies to generate behavior in new, unfamiliar problem situations. Obviously, this will often be the case if no task-specific, automated procedures are available (i.e., triggered by cues in the current situation). The use of analogy can best be conceptualized as a kind of mapping process (e.g., Anderson & Thompson, in press). As discussed in the previous section, students may use worked examples as a kind of *concrete* schemata to map their new solutions; in interpreting *cognitive* schemata, the key to the use of the schema is interpreting it by general procedures and mapping it onto the current knowledge of the situation to create a new solution (Hesketh, Andrews & Chandler, 1989). Thus, novices compare the current problem situation to information available in worked examples; with increasing expertise, the current problem situation can be compared with cognitive schemata retrieved from memory. As discussed before, such controlled processing has the advantage of flexibility, but it has the disadvantage that it works slowly and it may lead to errors due to processing overload.

If analogy repeatedly leads to the desired solutions, the schemata themselves may eventually be compiled into task-specific procedures that apply to particular classes of related problems and that directly produce the effect of the analogy without making reference to schemata. For instance, if a general design schema is repeatedly used to decompose a certain class of problems in subproblems in a process of stepwise refinement, the decomposition process for this class of problems may be automated. And likewise, if the application of a particular programming language template repeatedly leads to the desired solution for a certain class of subproblems, this template may be compiled into problem-specific procedures that are automatically applied when confronted with subsequent similar subproblems.

Schema acquisition and the design of practice. The assumption that the acquisition of schemata is often the result of mindful abstraction from concrete problems and their solutions has some important implications for the design of instruction. First, it is clear that the confrontation with a wide range of different problems and solutions to these problems, that will often have the form of actual computer programs or worked examples, is important to give inductive processes the opportunity to build, generalize, or specialize schemata. Obviously, mindful abstraction is not possible if there are no concrete objects, events, or activities to abstract away from. For instance, to develop a hierarchy of programming plans students must be confronted with a wide range of programs that demonstrate the use of programming language templates.

Second, there is evidence that mindful abstraction is an effortful process that requires the conscious attention of the learner. This leads to the additional implication that one should *provoke* this mindful decontextualization and generalization. Whereas Salomon and Perkins (1987) have stressed this point, they are not very specific about how exactly to provoke mindful abstraction in instructional materials. Instead, they focus on the role of the teacher, and remark that "... mindful abstraction is facilitated by a high teacher-student ratio, socratic interaction with the learners, and a great sensitivity on the part of the teacher for the ebb and flow of enthusiasm and understanding in the individual student..." (p. 164).

Schema acquisition and transfer. Acquired schemata may explain transfer by the presence of relevant knowledge from other problem solving situations and in particular, on how that knowledge is organized in schemata. Indeed, the quality of the induced schemata has been found to be highly predictive of subsequent transfer performance (e.g., Gick & Holyoak, 1983). In fact, this explanation is closely related to the gestalt theory of *structural understanding*, which claimed that transfer from one task to another is achieved by arranging learning situations so that a learner can gain insight into the problem to be solved. Bartlett (1932) first elaborated this view in his schema theory, which predicts that transfer will occur if one can relate the present problem to existing schemata, that is, to concepts and ideas in memory. Interpreting the selected schema and reorganizing the new situation according to this particular schema is, again, a form of controlled processing. Thus, both the acquisition of schemata and their use in transfer tasks requires effort and conscious attention from the learner. Salomon and Perkins (1987) refer to this transfer mechanism, which results from mindful abstraction from one situation and application to another, as *high road* transfer.

In increasingly further transfer within the programming domain, the overlap of task-specific procedures that were learned in the original task and are also applicable in the transfer task decreases. As a result, the availability of relevant schemata that may offer useful analogies becomes increasingly important in reaching further transfer (Jelsma, van Merriënboer, & Bijlstra, in press). A central empirical question concerns how these analogies are noticed and then applied to generate solutions to the new transfer problems. Spontaneously noticing the analogy is often a prerequisite for successful transfer in realistic problem situations. But, in learning schemata, it may help to explicitly state that a schema is also applicable in certain transfer situations; then, cues may be added to the schema that will facilitate its activation in these particular situations.

To summarize, automation and schema acquisition both play an important role in learning computer programming. The automation of procedures requires extensive varied practice, is facilitated by the availability of worked examples, and provides identical elements that may help to solve familiar aspects of new programming problems; furthermore, in solving a particular programming problem the availability of task-specific procedures frees up processing resources that may be devoted to various controlled processes. The acquisition of schemata, such as a general design schema or programming plans, requires mindful abstraction, presupposes the confrontation with a well-chosen range of problems and their solutions (i.e., worked examples), and provides analogies that may guide subsequent behavior in solving unfamiliar aspects of new programming problems.

TEACHING COMPUTER PROGRAMMING

In the previous section, it was argued that automation and schema acquisition make their own demands to the design of practice in teaching computer programming. With regard to these requirements, three instructional design principles can be formulated:

1. To facilitate automation, extensive varied practice should be provided;
2. To facilitate schema acquisition, mindful abstraction from examples should be provoked, and
3. To facilitate *both* automation and schema acquisition, worked examples should be directly available during practice.

All three instructional principles can be considered to be important to reach higher learning outcomes and near transfer. First, note that principles 1 and 2 need not be compatible. In teaching programming, one can give full priority to automation, full priority to schema acquisition, or priority to automation of certain aspects of the task and schema acquisition for other aspects of the task. In the following, the authors will advocate the *middle road* to transfer. Second, the authors will argue that principle 3, which claims that useful worked examples should always be available during practice, is largely neglected in current programming instruction and may—at least partly—explain low learning outcomes and, especially, the lack of ability to solve new programming problems (near transfer). Finally, this paper will discuss some more specific guidelines to provoke mindful abstraction, based on the assumption that a heavy emphasis on the use of worked examples is provided in the instruction.

The Middle Road to Transfer

One may stress the importance of automation and extensive varied practice in learning computer programming. But obviously, in elementary high school programming courses there is neither occasion for extensive practice (approximately. 100- 500 hours to reach mastery level, Anderson, 1982; Kurland, Mawby & Cahir, 1984), nor for much task variation within practice. Given this contradiction, for example de Corte and Verschaffel (1986) are moving away from a concern of programming. They argue that learning to program requires a time commitment that is out of the question in educational settings. Instead, they advocate a shift in attention to more constraining, quicker-to-learn general purpose application packages.

On the other hand, one may stress the role of schema acquisition and mindful abstraction. Salomon and Perkins (1987) argue, like de Corte and Verschaffel, that extensive varied practice to mastery or near automaticity, which they call the low road to transfer, is an inconvenient road to learning to program because automation is a slow process and the available time in school settings is severely limited. But instead, they argue that schema acquisition by mindful abstraction, which they call the high road to transfer, does offer opportunities to reach near transfer of programming skills and even to harvest general cognitive benefits from programming instruction. Obviously, students will stick to controlled processing when they generate their programs by interpreting schemata, that is, by using analogy; but in school settings this certainly is a more realistic goal to strive for than automation because schema acquisition may more rapidly occur.

Given the presented framework, the authors propose the middle road to transfer and claim that automation of the more familiar aspects of the programming task *is* of great importance in learning to program because it frees up processing resources that may be devoted to both the acquisition of n ew schemata and the interpretation of existing schemata. These processes are necessary to perform the unfamiliar aspects of the task. Thus, in solving a new programming problem the situation will be as follows. First, familiar aspects of the task (i.e., those aspects that are consistent over problem situations), such as proceeding in the programming environment, choosing the correct basic commands and applying syntactic rules can be performed by task-specific procedures. These procedures can be applied fast, without errors, and with little or no demands on processing capacity. Second, new aspects of the task can be solved by the use of analogy. Schemata such as programming plans (learned programming language templates) should be available to help to find a solution and these schemata can be interpreted thanks to processing resources that are freed up by automation of the more familiar aspects of the programming task.

In conclusion, automation should be seen as a process that is complementary to schema acquisition because it facilitates problem solving by analogy by freeing up the required cognitive resources. In addition, the authors fully subscribe that the provocation of mindful abstraction is important to reach schema acquisition and thus increase the possibility that near transfer effects occur. In other words, well-designed practice should provide extensive training of basic skills, but under strict conditions that vigorously promote the acquisition of schemata. In the following sections, the authors claim that a heavy use of worked examples during practice supports both automation and schema acquisition, and that the provocation of mindful abstraction is considerably simplified if such examples are the central component in programming instruction.

The Neglected Use of Worked Examples

The conventional way to teach computer programming is to present stereotyped sequences of instructional materials and problems. Typically, students are offered (a) a small amount of new programming language features along with some syntactic details, (b) a small number of illustrative problems and solutions in the form of computer programs that demonstrate the use of the new material, and (c) a relatively large number of programming problems for which students have to generate new computer programs.

From research in other domains, it is known that this traditional procedure used to enhance problem solving skills, that is, extensive practice on many conventional problems, is relatively ineffective. This evidence has been obtained both from solving puzzle problems (Mawer & Sweller, 1982; Sweller, 1983; Sweller & Levine, 1982; Sweller, Mawer & Howe, 1982) and from solving mathematical problems (Owen & Sweller, 1985; Sweller & Cooper, 1985; Sweller, Mawer & Ward, 1983). In particular, the lack of guidance and modelling during problem solving seems to impose a high processing load which may result in either directing the attention away from those aspects of the task that are important in learning or in completely losing ones way (Sweller, 1988). In addition, such ineffective practice and cognitive overload may eventually lead to decreased motivation and a further impairment of performance.

In the view of the authors, this undesirable situation can also be observed in most elementary computer programming courses. First, low learning outcomes and the lack of ability to solve new programming problems clearly demonstrate the ineffectiveness of current programming instruction. Second, high processing load during elementary computer programming and its negative effects on learning has been frequently reported (see, Anderson & Jeffries, 1985). And finally, the presentation of illustrative problems and their solutions in isolation from practice has been argued to be highly ineffective (Van Merriënboer & Krammer, 1987). During problem solving, students have to search for examples that fit in with their solution and they must turn back leaves, looking for examples analogous to the solution. This is a difficult task as students cannot be sure that a useful example is available; sometimes an example at first glance looks similar to the solution of the problem at hand but in fact it cannot be mapped correctly, which may result in serious mistakes.

Obviously, more effective practice should reduce processing load and redirect attention to those aspects of the task that facilitate learning. A heavy use of worked examples, which are directly available during practice and that provide a solution with a format that is similar to the format of the desired solution to the posed problem, may be the key to achieving this goal. Whereas, to the author's knowledge, no research is available in the field of computer programming, Cooper and Sweller (1987) reported that the simultaneous presentation of worked examples and problems in learning mathematical problem solving had a facilitating effect on automation as well as schema acquisition. In their experimental design, one group first received some illustrative examples and then a set of conventional problems; the other group received the same problems, but each problem was accompanied with an identical format problem that had the solution written out in a manner similar to that of the illustrative examples in the other group. The direct availability of useful worked examples during practice was found to be far more effective than the conventional use of illustrative examples. In particular, the use of worked examples shortened the acquisition phase, reduced the number of errors made during acquisition, and improved both near and far transfer performance.

Concluding, the main claim of this paper in regards to the design of practice in elementary computer programming is that during practice useful worked examples, which have the form of correct, well-structured programs with a format similar to that of the desired solution, should be directly available. As several other authors have argued on various grounds, the use of such worked examples has not been taken seriously enough in current programming instruction (e.g., Dalbey, Tourniaire, & Linn, 1985; Pea, 1986). In this respect, the authors fully agree with Lieberman (1986), who proposes that examples of correct solutions should be the *kernel* of well-designed practice. First, students can use such examples as blueprints to map their new solutions, which supports automation. And second, students may generalize from the examples to learn new programming principles, design techniques and, in particular, programming plans or programming language templates so that they also support schema acquisition. In addition, the direct coupling of practice with worked examples will make it easier for the teacher to articulate his or her expertise, because by presenting the students selected worked examples the—for the greater part—tacit knowledge, which is difficult to verbalize, may be implicitly conveyed. And finally, the use of worked examples during practice might simplify the provocation of mindful abstraction; at least, students are continuously confronted with materials they can abstract away from.

How to Provoke Mindful Abstraction?

In the previous section, the authors argued that a heavy use of worked examples during practice might be essential to reach more effective programming instruction. However, the presentation of worked examples alone will often not be sufficient, because the use of these examples as blueprints to map new solutions as well as the generalization of certain aspects of the examples requires the voluntary investment of effort, or the conscious attention, from the learner. In experimental studies, such as that of Cooper and Sweller (1987), learners will often be highly motivated and inclined to invest mental effort. However, in typical school settings one often has to deliberately provoke mindful abstraction. Thus, it is particularly important to focus on the question "how do we get the learners to thoroughly study the worked examples, and abstract away from their details?".

As a first observation, it should be obvious that a good teacher always points out what *is*, and what is *not* important about the worked examples. As Anderson, Boyle, Corbett and Lewis (1986) as well as Lieberman (1986) pointed out, the worked examples should be *annotated* with information about what they are supposed to illustrate. As programming plans, or learned programming language templates, are a particularly important kind of schemata to acquire during programming instruction (Ehrlich & Soloway, 1984; Soloway, 1985), it may be desirable to annotate the examples by explicitly referring to the newly-to-learn templates they use. In addition, one should explicitly state the situations to which the demonstrated principles or techniques might transfer to increase the chance that this transfer actually occurs (Gick & Holyoak, 1983). Only then, contextual cues may be added to the schemata that increase the likelihood that the appropriate schema is found and can be applied in the transfer situation.

A further, and in the opinion of the authors, particularly powerful route to provoke mindful abstraction is the use of programming assignments that require both the generation of code and a thorough study of worked examples. An obvious method to reach this goal is to confront the students with useful worked examples that have to be completed; the examples have the form of *incomplete*, but well-structured, understandable computer programs. Then, the students actually have to design and generate programming code and they extensively train basic skills such as proceeding in the programming environment, choosing the correct basic commands and applying syntactic rules, which is seen as essential to the acquisition of programming skill according to the proposed middle-road to transfer. But furthermore, the students are required to study the worked examples carefully because there is a direct, natural bond between examples and practice: They cannot correctly finish an assignment without understanding the program that has to be completed. Provided that the incomplete programs are well-chosen, that is, contain elements that are also relevant to completing them (e.g., the incomplete program contains a certain looping structure that is also necessary to correctly complete it), this method can be expected to facilitate both automation, because the students have blueprints available to map their new—partial—solutions, and schema acquisition, because they are forced to mindful abstraction from the incomplete programs.

In two studies that compared the effectiveness of these so-called "completion assignments" with conventional assignments (i.e., solving programming problems), higher learning outcomes for program construction tests were found for students who worked on the completion assignments (van Merri nboer, in press; van Merrië nboer & de Croock, 1989). The completion assignments that were used always consisted of a programming problem and a partial solution to this problem that was presented on-line, together with three components: (a) information on the language features and templates used in the partial solution and information on the situations in which those templates are useful, (b) questions on the structure and the workings of the incomplete program, and (c) task instructions to run, complete or change the incomplete program. Figure 1 presents a—shortened and simplified— example of a completion assignment to indicate its different components. The higher learning outcomes after working on the completion assignments could be well explained by a better application of programming language templates during the construction of new computer programs, indicating superior schema acquisition.

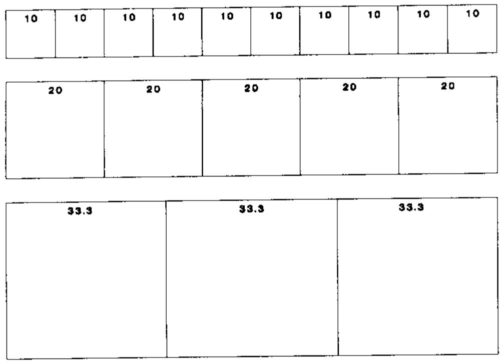
Finally, it should be noted that a heavy use of worked examples during practice in such a way that mindful abstraction is provoked places constraints on the sequence and variability of those examples. Whereas it goes beyond the scope of this article to elaborate these aspects, some important guidelines are the following. Obviously, a sequence of worked examples should start with simple examples and build to more complex examples. This will enable the students to initially acquire relatively simple schemata that are applicable to a wide range of problems, and subsequently develop more specialized schemata that may handle more exceptional cases. Furthermore, the sequence of worked examples should be varied in such a way that the differences and the similarities of one situation and solution to other, related situations and solutions is clarified to the students. For example, when and how to use a WHILE loop in a program should be clearly contrasted to when and how to use a REPEAT UNTIL. And to conclude, in the opinion of the authors it is self-evident that the amount of worked examples should be as high as possible, provided that they are directly coupled to practice and presented in such a way that they provoke mindful abstraction.

DISCUSSION

This paper offered a description of two learning processes that may be considered to be particularly important in learning elementary computer programming.

Problem Specification

Write a program to draw a series of a pre-specified number of squares. The total length of the series must be 100 units. Examples of desired output are:



Partial Solution

```
0010 PROC series100(number)
0020       right(90)
0030       FOR count1 := 1 TO number DO
0040             forward(100/number)
0050       ENDFOR count1
0060 ENDPROC series100
```

Information

The partial solution presents another example on the use of the FOR-loop. Remember that it should be used in case you want to repeat one or more program lines a *fixed* number of times:

- In previous programs, fixed meant a *constant* number of times (e.g., 4 or 65).
- In the presented partial solution, fixed means a number of times that is *pre-specified* as an argument to the procedure-call.

Question(s)

The procedure-call *series100(5)* on the partial solution yields:

a. one square with side lengths of 20 units
b. one square with side lengths of 100 units
c. one line with a length of 100 units
d. one line with a length of 500 units

Task Instruction(s)

1. Load the partial solution from disk and observe its output for different procedure-calls.
2. Complete the program to meet the given problem specification.

Figure 1. A simplified example of a completion assignment, using the programming language Comal-80 (Christensen, 1982).

Automation leads to task-specific procedures that may directly control programming behavior; schema acquisition offers cognitive structures, such as programming plans, that provide analogies in new problem situations. Based on the analysis of these learning processes, the authors argued that traditional practice, that is, solving many conventional programming problems after receiving some illustrative examples, is ineffective in terms of learning outcomes and transfer. Well-designed practice should provide extensive training of basic skills, but under strict conditions that vigorously promote the acquisition of schemata.

It was claimed that useful worked examples should always be available during practice, and that effortful, mindful abstraction from these examples should be provoked both by annotation of the examples and, in particular, by having the students to finish incomplete programs that concurrently serve as worked examples. These requirements to practice are easily met in a programming curriculum that may be labeled the *reading approach* (e.g., Deimel & Moffat, 1982). Four phases are distinguished in this curriculum. In the first phase, students run working programs, observe their behavior and evaluate their strengths and weaknesses. In the second phase, students are actually introduced to well-structured programs; their primary activities in this phase are reading and hand tracing of programs. During the third phase, students are confronted with completion assignments: They amplify and modify existing programs and practice both design and coding aspects on a modest scale. Only after students have reached a reasonable level of proficiency, they generate programs on their own and continue practicing basic design techniques and structured coding. For a comprehensive evaluation of this curriculum and its contrasts with other prevailing introductory programming curricula, see van Merriënboer & Krammer (1987).

Whereas the present article focussed on near transfer, or the ability to solve *new* programming problems, the framework has also some clear implications for reaching far transfer effects to other domains, that is, for the development of context independent strategies. In far transfer situations, the overlap of task-specific procedures with the original task heavily decreases so that the importance of acquired schemata increases (Jelsma et al., in press). Consequently, the provocation of mindful abstraction is particularly important to develop context independent strategies. In addition, it should explicitly be stated to the students that the learned techniques might transfer to an expressly named domain, so that cues become available in the acquired schemata that may activate them in the new context. For example, if one teaches students top-down design techniques in elementary programming, it should explicitly be stated that these techniques are also useful in writing an essay to be able to reach transfer to this domain.

But obviously, even as a top-down design schema has been acquired, and cues are available that may activate it when writing an essay, there is no warrant that the desired far transfer effects will actually occur. Just like schema acquisition, the interpretation of schemata is a controlled process that requires the investment of effort. As in near transfer, the learner will only be able to do so if the more familiar aspects of the task are automated; then, processing resources are available that may be devoted to mapping the top-down design schema to the new essay-writing context. If, on the other hand, students still have difficulties with writing grammatically correct sentences, the chance that transfer effects occur is low because no processing resources will be available to interpret the top-down design schema. And in addition, even as these cognitive resources *are* available, the students must be willing to invest effort or, simply stated, they must be willing to do their best to apply the learned skills in the new context. In conclusion, the proposed instructional principles are believed to increase the likelihood that—both near and far—transfer effects occur, but they can never guarantee such transfer because motivation eventually plays a crucial role in the learning and transfer of computer programming skills.

Finally, the authors are completely aware of the fact that the proposed approach to teaching elementary programming has far-reaching implications for designing elementary programming courses. Nowadays, there are almost no courses available which offer practice that is directly coupled to useful worked examples and explicitly provokes mindful abstraction from these examples. Much strenuous work will have to be done to design and implement such courses, because the good teaching of programming heavily depends on the laborious art of selecting and sequencing good examples to present to the students. However, given the disappointing learning outcomes and the lack of transfer effects of current programming instruction, this investment is believed to be worth while. It is the author's firm conviction that it is high time to seriously reconsider the traditional approach to teaching programming.

REFERENCES

Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review*, **89**, 369-406.

Anderson, J. R. (1983). *The architecture of cognition*. Cambridge: Harvard University Press.

Anderson, J. R. (1987). Skill acquisition: Compilation of weak-method problem solutions. *Psychological Review*, **94**, 192- 210.

Anderson, J. R., Boyle, C. F., Corbett, A., & Lewis, M. (1986). *Cognitive modelling and intelligent tutoring* (Tech. Rep. No. ONR-86-1). Pittsburgh, PA: Carnegie Mellon University.

Anderson, J. R., & Thompson, R. (in press). Use of analogy in a production system architecture. In S. Vosniadou & A. Ortony (Eds.), *Similarity and analogical reasoning*. Cambridge: Cambridge University Press.

Anderson, J. R. Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science*, **8**, 87-129.

Anderson, J.R., & Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from working memory. *Human-Computer Interaction*, **1**, 107-131.

Bartlett, F. C. (1932). *Remembering*. Cambridge, England: Cambridge University Press.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, **9**, 737-751.

Carbonell, J. G. (1984). Learning by analogy: Formulating and generalizing plans from past experience. In R. S. Michalsky, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. Vol. 1 (pp. 137-161). Berlin: Springer-Verlag.

Carbonell, J. G. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. S. Michalsky, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. Vol. 2 (pp. 371-392). Los Altos, CA: Morgan Kaufman Publishers.

Christensen, B. R. (1982). *Beginning Comal*. Chichester: Ellis Horwood.

Cooper, G., & Sweller, J. (1987). Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of Educational Psychology*, **4**, 347-362.

Dalbey, J., & Linn, M. C. (1985). The demands and requirements of computer programming: A literature review. *Journal of Educational Computing Research*, **1**, 253-274.

Dalbey, J., Tourniaire, F., & Linn, M. C. (1985). *Making programming instruction cognitively demanding: An intervention study (ACCCEL report).* Berkeley: University of California, Lawrence Hall of Science.

Deimel, L. E., & Moffat, D. V. (1982). A more analytical approach to teaching the introductory programming course. In J. Smith and M. Schuster (Eds.), *Proceedings of the NECC* (pp. 114-118). Columbia: The University of Missouri.

De Corte, E., & Verschaffel, L. (1986). Effects of computer experience on children's thinking skills. *Journal of Structural Learning*, **9**, 161-174.

DuBoulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, **2**, 57-73.

Ehrlich, K., & Soloway, E. (1984). An empirical investigation of the tacit plan knowledge in programming. In J. Thomas & M. L. Schneider (Eds.), *Human factors in computer systems* (pp. 113-133). Norwood, NJ: Ablex Publishing Corp.

Gick, M. L., & Holyoak, K. J. (1983). Schema induction and analogical transfer. *Cognitive Psychology*, **15**, 1-38.

Goodyear, P. (1987). Sources of difficulty in assessing the cognitive effects of learning to program. *Journal of Computer Assisted Learning*, **3**, 214-223.

Hesketh, B., Andrews, S., & Chandler, P. (1989). Opinion—Training for transferable skills: The role of examples and schema. *ETTI*, **26**, 156-165.

Jeffries, R., Turner, A. A., Polson, P. G., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 255-284). Hillsdale, NJ: Erlbaum Associates.

Jelsma, O., Van Merriënboer, J. J. G., & Bijlstra, J. P. (in press). The ADAPT design model: Towards instructional control of transfer. *Instructional Science.*

Kurland, D. M., Mawby, R., & Cahir, N. (1984). The development of programming expertise in adults and children. In M. Kurland (Ed.), *Developmental studies of computer programming skills* (Tech. Rep. 29). New York: Bank Street College.

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research*, **2**, 429-458.

Lieberman, H. (1986). An example based environment for beginning programmers. *Instructional Science*, **14**, 277-299.

Linn, M. C. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher*, **14**, 14-29.

Mawer, R., & Sweller, J. (1982). The effects of subgoal density and location on learning during problem solving. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, **8**, 252-259.

Mayer, R. E. (1981). The psychology of how novices learn computer programming. *Computing Surveys*, **13**, 121-141.

Mayer, R. E. (1982). Contributions of cognitive science and related research in learning to the design of computer literacy curricula. In R. Seidel, R. Anderson, & B. Hunter (Eds.), *Computer literacy* (pp. 129-159). New York: Academic Press.

Mayer, R. E. , & Greeno, J. G. (1972). Structural differences between learning outcomes produced by different instructional methods. *Journal of Educational Psychology*, **63**, 165-173.

McCoy Carver, S., & Klahr, D. (1986). Assessing children's LOGO debugging skills with a formal model. *Journal of Educational Computing Research*, **2**, 487-525.

Norman, D. A. (1981). Categorization of action slips. *Psychological Review*, **88**, 1-15.

Owen, E., & Sweller, J. (1985). What do students learn while solving mathematical problems? *Journal of Educational Psychology*, **77**, 272-284.

Pea, R. D. (1986). Language-independent conceptual 'bugs' in novice programming. *Journal of Educational Computing Research*, **2**, 25-36.

Pea, R. D., & Kurland, M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology*, **2**, 131-168.

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

Perkins, D. N., & Salomon, G. (1989). Are cognitive skills context-bound? *Educational Researcher*, **18**, 16-25.

Pirolli, P. L., & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*, **3**, 240-272.

Proctor, R. W., & Reeve, T. G. (1988). The acquisition of task-specific productions and modification of declarative representations in spatial-precueing tasks. *Journal of Experimental Psychology: General*, **117**, 182-196.

Putnam, R. T., Sleeman, D., Baxter, J. A., & Kuspa, L. K. (1986). A summary of misconceptions of high school BASIC programmers. *Journal of Educational Computing Research*, **2**, 459-471.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, **13**, 389-414.

Salomon, G., & Perkins, D. N. (1987). Transfer of cognitive skills from programming: When and how? *Journal of Educational Computing Research*, **3**, 149-169.

Schneider, W., & Shiffrin, R. M. (1977). Controlled and automatic human information processing: I. Detection, search, and attention. *Psychological Review*, **84**, 1-66.

Shiffrin, R. M., & Schneider, W. (1977). Controlled and automatic human information processing: II. Perceptual learning, automatic attending, and a general theory. *Psychological Review*, **84**, 127-190.

Singley, M. K., & Anderson, J. R. (1985). The transfer of text-editing skill. *International Journal of Man-Machine Studies*, **22**, 403-423.

Singley, M. K., & Anderson, J. R. (1988). A keystroke analysis of learning and transfer in text editing. *Human Computer Interaction*, **3**, 223-274.

Sleeman, D., Putnam, R. T., Baxter, J. A., & Kuspa, L. K. (1986). Pascal and high school students: A study of errors. *Journal of Educational Computing Research*, **2**, 5-24.

Soloway, E. (1985). From problems to programs via plans: The content and structure of knowledge for introductory LISP programming. *Journal of Educational Computing Research*, **1**, 157-172.

Sweller, J. (1983). Control mechanisms in problem solving. *Memory and Cognition*, **11**, 32-40.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, **12**, 257-285.

Sweller, J., & Cooper, G. A. (1985). The use of worked examples as a substitute for problem solving in algebra. *Cognition and Instruction*, **2**, 59-89.

Sweller, J., & Levine, M. (1982). Effects of goal specificity on means-ends analysis and learning. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, **8**, 463-474.

Sweller, J., Mawer, R., & Howe, W. (1982). Consequences of history-cued and means-ends strategies in problem solving. *American Journal of Psychology*, **95**, 455-483.

Sweller, J., Mawer, R., & Ward, M. (1983). Development of expertise in mathematical problem solving. *Journal of Experimental Psychology: General*, **112**, 634-656.

Thorndike, E. L., & Woodworth, R. S. (1901). The influence of improvement in one mental function upon the efficiency of other functions. *Psychological Review*, **8**, 247-261.

Van Merriënboer, J. J. G. (in press). Strategies for programming instruction in high school: Program completion vs. program generation. *Journal of Educational Computing Research*, **6**.

Van Merriënboer, J. J. G., & De Croock, M. B. M. (1989, September). *Strategies for computer-based programming instruction: Program completion vs. program generation.* Paper presented at the Third European Conference for Research on Learning and Instruction (EARLI), Madrid, Spain.

Van Merriënboer, J. J. G., & Krammer, H. P. M. (1987). Instructional strategies and tactics for the design of introductory computer programming courses in high school. *Instructional Science*, **16**, 251-285.