Instructional Science 16: 251-285 (1987)
© Martinus Nijhoff Publishers (Kluwer), Dordrecht – Printed in the Netherlands

Instructional strategies and tactics for the design of introductory computer programming courses in high school

JEROEN J. G. VAN MERRIENBOER & HEIN P. M. KRAMMER
Department of Instructional Technology, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract. This article offers an examination of instructional strategies and tactics for the design of introductory computer programming courses in high school. We distinguish the Expert, Spiral and Reading approach as groups of instructional strategies that mainly differ in their general design plan to control students' processing load. In order, they emphasize top-down program design, incremental learning, and program modification and amplification. In contrast, tactics are specific design plans that prescribe methods to reach desired learning outcomes under given circumstances. Based on ACT* (Anderson, 1983) and relevant research, we distinguish between declarative and procedural instruction and present six tactics which can be used both to design courses and to evaluate strategies. Three tactics for declarative instruction involve concrete computer models, programming plans and design diagrams; three tactics for procedural instruction involve worked-out examples, practice of basic cognitive skills and task variation. In our evaluation of groups of instructional strategies, the Reading approach has been found to be superior to the Expert and Spiral approaches.

Contents

1 Introduction

Computer science has recently become an integral part of the high school curriculum in most Western countries. Typically, these computer science courses involve elementary computer programming to a large extent because some programming knowledge is generally seen as a necessary element of modern literacy; in addition, programming is usually expected to foster the development of specific cognitive skills which may positively affect problem solving behavior in other school disciplines (Clements, 1986a, 1986b, 1987; Clements and Gullo, 1984; Ehrlich, Soloway and Abbott, 1982; Schulz-Zander, 1986; Soloway, Lochhead and Clement, 1982).

This article deals with the question of what we may learn from cognitive psychology for the instructional design of introductory programming courses. Leading research concerning cognitive psychology and elementary computer programming concentrates on

1 computer models (Mayer, 1979, 1981, 1982; Mayer and Bromage, 1980)

2 programming languages (Baron, Szymanski, Lock and Prywes, 1985; Green, 1983; Green, Sime and Fitter, 1980; Samurçay, 1985; Soloway, Bonar and Ehrlich, 1982)

3 programming plans (Ehrlich and Soloway, 1984; Soloway, 1985; Spohrer, Soloway and Pope, 1985)

4 general design models (Dershem, 1980; Hoc, 1981; Linn, 1985; Mandinach and Linn, 1986)

5 misconceptions (Bayman and Mayer, 1983; Bonar and Soloway, 1985; Sleeman, Putnam, Baxter and Kuspa, 1986; Soloway, Ehrlich, Bonar and Greenspan, 1982)

6 group work (Webb, 1983, 1984; Webb, Ender and Lewis, 1986)

7 graphical representations of algorithms (Brooke and Duncan, 1980; Fitter and Green, 1979; Green, 1982; Sheppard, Kruesi and Curtis, 1981; Shneiderman, Mayer, McKay and Heller, 1977)

8 intelligent tutoring systems (Anderson and Reiser, 1985; Anderson and Skwarecki, 1986; Johnson and Soloway, 1985a, 1985b; Reiser, Anderson and Farrell, 1985).

The goal of this paper is the presentation of instructional design principles for introductory programming courses in high school. We distinguish between instruction and management as two different aspects of teaching. Whereas principles of management depend on class size, principles of instruction hold irrespective of the number of students in the class. In this article we confine ourselves to the instructional aspects of teaching. For instance, the arrangement of group work in programming courses remains beyond the scope of this study.

In addition, we distinguish between instructional *strategies* and instructional *tactics*. Instructional strategies are general design plans that mainly differ in their control of students' processing load but that all pursue the same high-level goals which teachers consider desirable; instructional tactics are specific design plans that prescribe methods to reach desired learning outcomes under given circumstances. As will be argued, an evaluation of strategies is, to a certain degree, possible by establishing the facility of applying tactics in those strategies. Thus, tactics may be used both to design introductory programming courses and to evaluate strategies that characterize existing courses.

The structure of our discourse is as follows. In the second section, a description will be given of three groups of instructional strategies that we call the *Expert* approach, the *Spiral* approach and the *Reading* approach. The Expert approach is a top-down approach, which starts the novice off with a complex but intrinsically motivating programming problem; the Spiral approach is a parallel syntax and semantic acquisition emphasizing small incremental steps and building up a program by mastering the basics (language constructs) first; the Reading approach recommends that students begin by understanding relatively complex existing programs and then modifying and enhancing those programs. In the third section, we will make a distinction between *declarative* instruction and *procedural* instruction. Based on ACT* (J. R. Anderson, 1983) and relevant research, six instructional tactics will be presented. Three tactics for declarative instruction involve concrete computer models, programming plans and design diagrams; three tactics for procedural instruction involve worked-out examples, basic skills and task variation. The fourth section includes an evaluation of the groups of instructional strategies, in which the Reading approach is found to be superior to other approaches. Finally, we will offer a discussion of our framework for instructional design in elementary programming and its research implications.

2 An overview of instructional strategies

Typically, introductory programming courses aim at four categories of skills (*e.g.*, Mayer, 1975; Webb, 1984). First, students have to learn how to proceed in the programming environment, which includes the operating system and the editor. Second, they should be able to apply the syntactic rules of the programming language. Third, they should gain understanding of the semantics of the language and master abilities that are necessary to analyze code so as to comprehend programs. Finally, students should become able to generate programs themselves. The first two categories are never seen as ultimate goals: Rather, they function as vehicles to reach the latter two categories of goals.

In elementary programming, time is lacking to learn all four categories of skills to the same high level of performance. In agreement with Anderson (1982), we distinguish three stages in the development of a complex cognitive skill. In the first stage, there is an *initial* performance level which enables students to generate the desired behavior to some degree, but their performance is slow and subject to many errors; in the second stage, there is an *intermediate* performance level which enables students to perform the skill faster and with most errors eliminated; in the third stage, there is a *high* performance level which enables students to perform the skill fast and without errors. In elementary programming, skills necessary to proceed in the programming environment and to apply syntactic rules have to be learned from an intermediate to a high performance level because they serve as vehicles to reach the ultimate goals. Skills necessary to comprehend or generate programs can only be acquired from an initial to intermediate performance level because available time does not allow the setting of higher instructional goals.

Two aspects can be distinguished in the ability to generate a program that is a possible solution to a given problem. First, students analyze the problem and produce an algorithmic solution; second, they translate this algorithm into program code. This distinction between *design* aspects and *coding* aspects in program generation is found in most curriculum documents and point-of-view papers on elementary programming (*e.g.*, Balzert and Hille, 1980; Schulz-Zander, 1981; Woodhouse, 1983).

In the 1970s, programming was presented as a fundamentally easy task: One only had to learn a programming language to engage in it. Programming courses were, in fact, language courses fully stressing the coding aspects of program generation. In the last decade, program generation came to be emphasized as a problem solving activity which included substantial design aspects.

This modern view often appeared difficult to implement. Students are generally inclined to rush to the computer and focus on the coding aspects of program generation. They frequently attempt to go from a detail of the problem specification to the programming code, without any consideration of how to solve the problem as a whole, how to plan the solution, or how to design the code (Dalbey and Linn, 1986). Thus, despite the resolution to stress careful design, the teacher is frustrated by the students' tendency to focus on the computer and the more concrete aspects of program generation.

Several instructional strategies were developed to overcome this difficulty. In this article we distinguish between three widespread groups of strategies: the Expert approach, the Spiral approach and the Reading approach. Our classification is based upon a literature search as well as on an investigation of existing courses and programming textbooks. The classification has been chosen to highlight the *differences* between approaches. As a consequence, our descriptions may – to a certain degree – be seen as exaggerations.

2.1 The Expert approach

The Expert approach in designing introductory programming courses is closely related to the ideas that originated within the discipline of structured programming (Dahl, Dijkstra and Hoare, 1972; Wirth, 1974; Yourdon, 1975). Originally, this discipline addressed itself to professional programmers. However, many ideas were embraced by teachers in elementary programming.

For over a decade, the discipline of structured programming has been having an enormous influence on the practice of programming because it urges programmers to work within certain limitations and according to certain rules. These constraints are claimed to ease the programming process for the programmer and to make programs easier to comprehend, to write and to debug. In agreement with Sime, Arblaster and Green (1977), we distinguish between constraints on the programming language and constraints on the programmer's procedure.

Language constraints

Interest in specific language features seriously began when Dijkstra (1968) assailed the GOTO statement. He argued that this statement made it difficult both to comprehend and to develop programs. The structured programming discipline called for more restriction in specifying control and data structures to be included in future languages. Ever since, much research has been done that demonstrated the value of structured language features in programming (see for a review, Curtis, 1983).

Several authors (*e.g.*, Atherton, 1981; Woodhouse, 1983) advocated the use of structured languages such as Pascal, PL/1, or COMAL-80 in introductory programming courses. They particularly criticized the use of BASIC for its lack of structure: Learning BASIC as a first programming language was supposed to interfere with subsequently learning a structured programming approach (*e.g.*, Atherton, 1982; Baird, 1982). Their opinion is summarized in the well-known statement: "BASIC damages the brain!".

Nowadays, the use of structured programming languages in introductory programming courses is commonly accepted. Thus, using a well-structured programming language is not reserved to the Expert approach but is possible and, in our opinion, highly desirable in all instructional strategies. Throughout this article, we presuppose the use of a structured programming language in the Expert, Spiral, as well as Reading approach: They do not differ in this respect.

Procedure constraints

Programming should essentially be seen as a problem solving activity in which the responsibility of the programmer extends beyond producing code that works. First, programmers have to structure their programs so that they are readable for others: Joni and Soloway (1986) referred to these constraints as "discourse rules".

Second, programmers are recommended to adopt a model of "stepwise refinement" to achieve algorithm and program design in a top-down fashion. Thus, they should begin writing a program by first specifying the top levels of the task-hierarchy to design the algorithm and then proceed to successively specify lower levels until the actual language code has been reached. Especially well-structured languages should make a top-down approach feasible because they facilitate breaking down large programming tasks into smaller subtasks.

In the design of introductory programming courses, the characteristic feature of the Expert approach is its emphasis on both algorithm and program design in a systematic top-down fashion. For this reason, students are offered problem specifications during the course that are characterized as non-trivial design problems. That is, from the outset of the course they receive problems for which algorithms have to be developed. Working according to the presented model of stepwise refinement should allow the students to concentrate more on the semantic content of the algorithm because less attention is required to track actions on lower program code levels.

2.2 The Spiral approach

This approach is closely related to the ideas of Ausubel (1968). Shneiderman (1977a) coupled Ausubel's educational theory to his syntactic/semantic model (Shneiderman and Mayer, 1979) of programmer behavior. In this model, syntactic knowledge is defined as unorganized knowledge of low level details, such as the syntax of language features and the names and arguments of functions; in contrast, semantic knowledge is defined as hierarchically organized knowledge with concepts ranging from lower levels, such as the assignment statement, to higher levels, such as the pattern of code for finding the mean of an array. Based on this distinction and on Ausubel's notion of "anchoring" new material to an "ideational structure" through a process of "progressive differentiation", Shneiderman (1977a) presented the Spiral approach for teaching introductory programming courses. According to him: "The Spiral approach is the parallel acquisition of syntactic and semantic knowledge in a sequence which provokes student interest by using meaningful examples, builds on previous knowledge, is in harmony with the student's cognitive skills, provides reinforcement of recently acquired material and develops confidence through successful accomplishment of increasingly difficult tasks" (p.193).

In short, more complex forms of knowledge are developed in a hierarchical manner. By selecting small instruction steps, complex ideas are built from combinations of simpler ideas. Each step must (a) contain both syntactic and semantic elements, (b) present a minimal extension of previous knowledge, (c) be explained in relation to former knowledge and (d) be trained in exercises. Syntactic knowledge is learned by repetition and must frequently be rehearsed to prevent forget-

ting; semantic knowledge is acquired through meaningful learning and is better resistant to forgetting. However, semantic information must always be presented in small units that are higher level organizations of previously acquired knowledge.

The characteristic feature of the Spiral approach is its emphasis on stepwise incremental learning. Problem specifications that are presented to students during the course gradually become more complex in both the coding and the design aspects that they require. Consequently, in the beginning of the course students receive more or less trivial problems that emphasize syntactic and lower level semantic knowledge. After students have gained more experience, problems become more complex and can be seen as non-trivial design problems that require serious algorithm design.

2.3 The Reading approach

This approach emphasizes the reading, modification and amplification of non-trivial, well-designed and working programs. Deimel and Moffat (1982) promoted the Reading approach and separated four phases in introductory programming courses. In the first – short – phase, students run working programs, observe their behavior and evaluate their strengths and weaknesses. In phase two, students are actually introduced to well-structured programs. Their primary activities in this phase are reading and hand tracing of programs. Thus, learning the specific language is largely done by extracting the language features from concrete programs. During the third phase, students modify and amplify existing programs and practice both design and coding aspects on a modest scale. Finally, students generate programs on their own and continue practicing basic design techniques and structured coding.

Several authors recommended strategies that we assign to the Reading approach. Dalbey, Tourniaire and Linn (1985) reported that students showed a serious lack of planning in program generation. They suggested that "it would seem quite appropriate to begin instruction with comprehension of program code... Those programs would demonstrate how planning is used in programming... Thus, students would have a better understanding of the role of planning in programming" (p. 18). Pea (1986) reported negative effects of "bugs" on the learning process in programming. Bugs are misconceptions that students have about the operation of computers and the working of programming languages. These misconceptions cause systematic errors in program comprehension and generation. According to Pea, "bugs like these could be snared if one used program *reading* or debugging activities as central components of programming instruction" (p. 34).

The characteristic feature of the Reading approach is its emphasis on program comprehension, modification and amplification. For this reason, students are confronted with non-trivial design problems from the beginning of the course.

However, these problems are presented in combination with their complete or partial solutions in the form of well-designed, well-structured and well-documented programs. The students' tasks gradually become more complex during the course, changing from using and analyzing programs, through modifying and extending programs, to independently designing and coding programs.

2.4 A preliminary comparison of strategies

In this section, we offer a preliminary overview of major similarities and differences between the Expert, Spiral and Reading approaches. A more thorough comparison and evaluation of instructional strategies ensues in a later section. Both the Expert approach and the Spiral approach employ program generation as a primary student activity. During the course, the complexity of presented problems gradually increases in both approaches. However, the Expert approach emphasizes top-down design aspects by immediately offering non-trivial design problems and top-down design techniques; the Spiral approach stresses parallel, stepwise teaching of syntactic and semantic aspects by offering relatively simple coding problems in the beginning of the course and more complex problems, demanding serious algorithm design, only later in the course.

Like the Expert approach, the Reading approach advocates the presentation of design problems in an early stage of the course. However, the Reading approach is taking a different route by not merely presenting problems but also complete or partial solutions in the form of well-designed programs: The complexity of offered problems is relatively constant throughout the course, whereas the students' tasks vary from comprehension, through modification and amplification, to designing and coding complete programs.

The most conspicuous differences occur between the Spiral approach and the Reading approach. Whereas the Spiral approach adopts program generation as a primary and, throughout the course, constant activity, the Reading approach employs the comprehension, modification and amplification of programs as subsequent student activities. Besides, problem complexity in the Spiral approach increases from relatively simple coding problems to more difficult design problems; in the Reading approach, problem complexity is high from the outset but the problems are presented in combination with their complete or partial solutions.

3 An overview of instructional tactics

Instructional *principles* can be formulated in a circumstances-method-outcomes format (Reigeluth, 1983). Circumstances are factors that influence the effects of methods – and are therefore important for prescribing methods – but they cannot

be manipulated; methods are manipulations to achieve different outcomes under different circumstances; outcomes are effects that provide a measure of alternative methods under different circumstances. Outcomes may be desired or actual: In this article, we are concerned with desired outcomes that are equivalent to instructional goals. For this reason, we formulate – instead of principles – instructional *tactics* in a goals-circumstances-method format. The structure of a tactic may be illustrated by an example of the classroom questioning behavior of teachers. Research on this topic suggests that the cognitive level of questions asked should depend on student home background and age (Gall, 1984). The following tactic could be derived from this research:

Example of a Tactic

GOAL(S):

- mastery of elementary skills

CIRCUMSTANCE(S):

- the teacher is engaged in a classroom discussion
- the age of the students is below eight years
- the students have disadvantaged home backgrounds

METHOD:

- ask mainly factual questions that students are expected to answer correctly

Each instructional tactic should have at least one goal, often one or more circumstances to delimit its validity and exactly one method. For our purposes, goals are formulated as desired learning outcomes; circumstances are formulated as characteristics of the student population and the subject matter; methods are formulated as manipulations affecting the instructional design. We have already distinguished four categories of desired learning outcomes for elementary programming: skills necessary to (a) proceed in the programming environment, (b) apply syntactic rules, (c) comprehend programs and (d) generate programs. In addition, these skills may be mastered up to either an initial, intermediate, or high performance level. In this section, we present six tactics that include goals, circumstances and methods in the above-mentioned format. Moreover, these tactics will be related to a common cognitive-psychological background and, when possible, they will be supported by relevant research.

3.1 Theoretical background

The ACT* theory ("Adaptive Control of Thought"; Anderson, 1982, 1983; Anderson, Greeno, Kline and Neves, 1981; Anderson, Kline and Beasly, 1980; Neves and Anderson, 1982) offers a suitable theoretical background for discussing instructional tactics in elementary programming because (a) it makes general

claims about the organization and acquisition of complex cognitive skills and it is capable of explaining most results in cognitive research on computer programming, (b) it offers various points of contact for instructional design as shown in the development of computer-based LISP tutors (Anderson and Reiser, 1985; Anderson and Skwarecki, 1986; Reiser, Anderson and Farrell, 1985), and (c) it has already been successfully applied to simulate learning processes involved in programming: In particular, these processes are simulated in GRAPES ("Goal-Restricted Production System"; Anderson, Farrel and Sauers, 1984), which embodies parts of the cognitive architecture as specified in ACT*.

Cognitive architecture

Newell and Simon (1972) promoted production system theories by making the claim that a set of condition-action pairs called "productions" underlies human cognition. The condition part specifies various features; if elements that match those features are in working memory, the production applies. The action part specifies what to do if the condition is matched; if the production applies, it adds new elements to working memory. As an extension of this cognitive architecture, ACT* makes a fundamental distinction between declarative and procedural knowledge; in addition to the active part of information in working memory and a set of productions in procedural memory, it contains a set of facts in declarative memory.

Declarative memory contains cognitive units to encode sets of elements that have a particular relationship. Cognitive units appear as elements of one another to create complex hierarchical structures: Interconnections between these structures and elements form a network. An activation process that is working on this network defines the working memory. Furthermore, each cognitive unit has a strength associated with it that is a function of the frequency of use; this strength determines the spread of activation throughout the network.

Working memory refers to active information that is basically declarative in nature. Each element that enters into working memory is a temporary source of activation. In addition, a single goal element may serve as a permanent source of activation. In particular, working memory contains temporary structures that are either created by the perception of objects in the outside world or deposited in working memory by actions of productions. If a structure is created in working memory, there is a probability that a permanent copy of it will be made in declarative memory; if a copy already exists, its strength will increase.

Procedural memory contains productions of which the conditions are matched with the structures in working memory. When the match is successful, temporary structures are added to working memory by execution of the action. Thus, productions operate on declarative knowledge that is currently active in working memory. In addition, a strengthening process increases the strength of a production with every successful application.

Goal-directed processing

In ACT*, special attention is given to goal-directed productions that match for a single goal in their condition as well as for other features. If the goal specification matches the current goal in working memory, it is given precedence over all other productions. The GRAPES system (Anderson *et al.*, 1984) restricts itself to such goal-directed productions to simulate the processes involved in learning to program. In fact, productions can create a goal structure that reflects the problem solver's plan of action. When a goal has been achieved, it is removed from working memory and attention is shifted to the next (sub)goal in the structure. For instance, a condition specifies a particular programming goal and some problem specifications; if the condition matches the contents of working memory, execution of the action may set new (sub)goals, reformulate the problem specification, or write program code.

Elaborative processing

According to ACT*, to-be-learned information always comes in declarative form. One of the best ways to increase students' memory for new information is to have them elaborate on the instructional material. Elaborative processing indicates that productions use declarative knowledge structures that already exist in memory to generate elaborations which are embellishments of the instructional material. The retrieved knowledge is called a *schema* because it provides a cognitive structure for understanding a situation in general terms. In the elaborative process, productions connect the schema with the instruction and infer information from the schema that is not in the instruction. As a result, the elaboration of the instructional material is a more richly connected cognitive structure than was specified in the instructional material. The processes involved in elaborative processing may be seen as a form of meaningful learning because subjects connect new material with one or more schemata that already exist in memory. These schemata provide structural understanding and may subsequently guide problem solving behavior.

Skill acquisition

Learning a complex cognitive skill develops from a declarative to a procedural stage by a process referred to as knowledge compilation. In the declarative stage students receive instruction about the skill mainly by reading textbooks and listening to lectures. New facts are stored in declarative memory. To generate behavior on the basis of newly acquired knowledge, students must use existing domain-independent productions to *interpret* those facts. Although the interpretation of knowledge in declarative form has the advantage of flexibility, it also has serious costs. The process is slow because interpretation requires continuous retrieval of facts from declarative memory and because the individual interpretative production steps are usually small.

Knowledge compilation creates task-specific productions through *practice*. It includes the subprocesses composition and proceduralization. Composition collapses sequences of productions into single productions and considerably speeds up production application because the new productions embody sequences of steps that are needed in a particular domain. Proceduralization embeds factual, task-specific knowledge in productions and reduces load on working memory because declarative information need no longer be held active. Hence, with practice the declarative knowledge is gradually converted into a procedural form in which it directly controls behavior. During knowledge compilation, the skill is performed at an intermediate level.

In the procedural stage, the performance level is high because knowledge about the skill is directly embodied in task-specific productions which may be applied very fast and with low demands on working memory. A further *tuning* makes the knowledge more selective in its range of applications. In ACT*, tuning includes – in addition to strengthening – generalization to create more general productions and discrimination to create more specific productions. However, it should be noted that in the PUPS successor of ACT* (PenUltimate Production System; Anderson, Boyle, Corbett and Lewis, 1986) there are neither generalization nor discrimination mechanisms that automatically compare the current situation with past situations. Whereas in ACT* generalizations and discriminations are productions rendered by automatic learning mechanisms, in PUPS they are schema-like, declarative knowledge structures that are produced by problem solving productions.

Our review of relevant aspects of ACT* set up a framework for discussing instructional tactics. We distinguish between declarative instruction and procedural instruction. Declarative instruction involves methods for the initial presentation of information about the computer, the programming language and the design process to facilitate the storage of new declarative knowledge; procedural instruction involves methods for the design of practice to facilitate knowledge compilation.

3.2 Tactics for Declarative instruction

Some general recommendations for declarative instruction, not limited to elementary programming, are: (a) let students explain new information in their own words, verbally or by taking notes, to relate the instructions to existing knowledge, (b) assess students' misconceptions about the task and subsequently use them in the design of instructional materials and (c) teach students not only useful actions but also the conditions under which those actions are useful (Larkin, 1979; Simon, 1980). These recommendations may eventually be specified to fulfil the conditions that we laid down for instructional tactics. However, in this article we limit ourselves to tactics for declarative instruction that have already been especially developed for elementary programming.

A problem with most instructional materials for elementary programming is that many things that students have to know are omitted; they must figure them out by trial and error. In our opinion, a key aspect for declarative instruction should be the teaching of facts, such as language statements and syntactic rules, in combination with schema-like knowledge to encourage elaborative processing. We will discuss three tactics that concern such schema-like knowledge by focussing on (a) the machine, by introducing a concrete computer model, (b) the programs, by offering programming plans and (c) the design process, by presenting a design diagram. These tactics are explicitly concerned with the acquisition of declarative knowledge so that their instructional goals are limited to mastery of skills at an *initial* performance level. Whereas it is obvious that ultimate goals involve higher performance levels, these are not within reach because practice is not yet included in declarative instruction.

Concrete computer models

DuBoulay, O'Shea and Monk (1981) introduced the distinction between a "black box approach" and a "glass box approach" in elementary programming. In the black box approach, students have no idea of what goes on inside the computer because they lack an adequate model. In the glass box approach, students do have such an idea because the instruction includes a concrete but simplified computer model. This model makes it possible to emphasize a "notional machine" on both a general level, such as in teaching the relationship between the terminal and the computer and a specific level, such as in teaching assignment statements (DuBoulay, 1986).

Mayer (1975) either gave students in an introductory BASIC course a concrete computer model or he did not. The group that received the model excelled in comprehension and generation of *new* programs; the group that received no model performed equally well on problems that were very much like the material in the instructional text. According to Mayer, the presentation of the model provided a context in which students could relate new instructions to an already familiar analogy. Consequently, the instruction resulted in a broader learning outcome.

In subsequent studies (Mayer, 1976; Mayer and Bromage, 1980), the model was presented either before or after the reading of the instructional text. In agreement with the previous results, the group that received the model before reading the text excelled in comprehension and generation of new programs as well as on recall of information that could be conceptually related to the operation of the computer. Thus, the model facilitated learning only if it was available to students before reading the instructional text.

Summing up, students who received a computer model before reading showed more integrated learning of information, which improved their comprehension and generation of new programs. We suppose that the presentation of the model early in the learning process encouraged elaborative processing. That is, the subsequent

reading of the instructional text resulted in a more richly connected knowledge structure, which improved program comprehension and program generation learning outcomes. As a consequence, our first instructional tactic for declarative instruction in elementary computer programming is:

Computer model tactic

GOAL(S):

- initial performance level in program comprehension and generation

CIRCUMSTANCE(S):

- students are pre-novices in computer programming and are in the declarative stage

METHOD:

- present a concrete computer model early in learning

Programming plans

Expert programmers at a glance recall far more information from a computer program than novices (McKeithen, Reitman, Rueter and Hirtle, 1981). Experts have more knowledge concerning programs and – perhaps even more important – this knowledge is better organized into cognitive structures. Adelson (1981) studied differences between expert and novice programmers in their recall of complete programs. Novices were attending to the syntactic surface structure of single program lines; experts used a more abstract hierarchical organization based on the functional principles in blocks of related program lines. In agreement with these results, Shneiderman (1976, 1977b) and Barfield (1986) found that experienced programmers could recall more lines of programming code than novices when the program was organized in executable order; however, the groups performed at similar levels when the programs consisted of random lines of code. Thus, experts seem to organize their knowledge of programs into cognitive structures that contain *templates* of language code.

Experts not only use such cognitive structures to comprehend programs but also to understand problem specifications in program generation. Atwood, Turner, Ramsey, Hooper and Sidorsky (1977) presented programming problems that subjects had to summarize in their own words. Whereas novices omitted all details of the problem specification, intermediates and experts emphasized the details that were of importance to program design. This is in agreement with the results of a study by Weiser and Shertz (1983), in which novices and experts had to sort programming problems. Novices sorted problems according to their field of application, such as word processing, data management and robotics; experts sorted problems regarding their deep structure, such as the fundamental algorithms that were underlying their solutions.

Ehrlich and Soloway (1984) presented a theory of programming plans in which they tried to identify the content of cognitive structures as used in programming. The goal of the theory is to improve the teaching of elementary programming and to support the building of computer based programming tutors (Johnson and Soloway, 1985a, 1985b). Fundamental in their approach is that expert programmers organize their programming knowledge into schemata that represent patterns of code that are associated with specific programming problems. Programming plans give a concrete form to these schemata as templates of programming code in combination with comments that describe the goals and reasons for the various expressions in the template. Such programming plans are largely independent of the programming language that is used and can be learned directly from instruction.

According to Soloway (1985), instruction should not only emphasize the syntax and statements of a particular programming language but also programming plans. This makes it possible to stress the structure of, and relationships between, specific programming problems and programs. Thus, the explicit presentation of programming plans supports the development of cognitive structures that are used in both the comprehension of programs and the understanding of problem specifications in program generation. In fact, programming plans begin to appear in textbooks that teach elementary programming (*e.g.*, Cooper and Clancy, 1982; Dale and Orschalick, 1983). Based upon the theory of programming plans, we formulate our second instructional tactic for declarative instruction:

Programming-plans tactic

GOAL(S):

- initial performance level in program comprehension and generation

CIRCUMSTANCE(S):

- students are novices in computer programming and are in the declarative stage

METHOD:

- explicitly present programming plans

Design diagrams

Jeffries, Turner, Polson and Atwood (1981) studied the processes involved in program design and distinguished three major mechanisms: (a) the decomposition of the problem specification into a collection of modules, (b) the specification of the relationships and interactions among modules as control structures that indicate when and under which conditions modules are activated and (c) the specification of data structures that are involved in the solution. Expert programmers have abstract knowledge concerning the processes involved in generating a good design and its overall structure. This knowledge is referred to as a general design schema.

The design schema may be used recursively to generate a decomposition of the problem into more and more detailed modules in a process of "successive 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 subproblem. This description of expert programming behavior is in accordance with both the dominant view of planning as a process that starts with high-level goals and refines them into achievable actions (Newell and Simon, 1972; Sacerdoti, 1977) and the principles of goal-directed processing as specified in ACT*. In addition, it is nearly equivalent to the model of stepwise refinement as promoted by the structured programming discipline.

In contrast to experts, novices do not possess a general design schema and usually have severe difficulties in coordinating their activities. Because they lack a structure for organizing their behavior, they are often unable to decompose the problem into appropriate subproblems, to correctly interface modules and to identify necessary data structures. In our opinion, a common difficulty in introductory programming courses is that they do not embody instructional tactics that reflect the students' need for "direction" in problem solving. Consequently, novices often attempt to go from their incomplete design to implementing the program, without further consideration of how to plan the complete solution. This often results in badly structured, buggy programs.

Bradley (1985) reported a positive correlation between top-down processing styles and learning outcomes in a 15-lesson introductory LOGO-course. Instructional materials may support such top-down processing by explicitly presenting a design diagram: A flow-chart or structured diagram prescribing in detail the actions and methods that ensure a systematic and effective design process. Equivalent design diagrams for solving elementary science problems are sometimes referred to as SAP-charts ("Systematic Approach to Problem-solving"; see for an example, Mettes, Pilot and Roossink, 1981). The presentation of a design diagram clarifies the complementary processes of successive refinement and top-down program design and facilitates the development of a general design schema. Based upon this idea, we formulate our last tactic for declarative instruction:

Design diagram tactic

GOAL(S):

- initial performance level in top-down program design and successive refinement

CIRCUMSTANCE(S):

- students are novices in computer programming and are in the declarative stage

METHOD:

- explicitly present a design diagram

3.3 Tactics for Procedural instruction

Whereas the key aspect in declarative instruction is teaching schema-like knowledge to encourage elaborative processing in the learning of related facts, the key aspect in procedural instruction is supporting processes involved in knowledge compilation and tuning. Thus, procedural instruction primarily involves the instructional design of practice to make the transition from the declarative stage to the procedural stage as smooth as possible. Note that practice does not imply that students always have the disposal of a computer; for instance, algorithm design may be practiced without a computer.

In general, students initially have serious difficulties in applying newly acquired declarative knowledge in practice. This is in agreement with ACT*, which predicts that in the declarative stage the use of knowledge by interpretative productions is – as a result of high demands on working memory – a slow process characterized by many errors. Three instructional tactics for procedural instruction involve (a) worked-out examples, (b) practice of basic cognitive skills and (c) task variation. Because these tactics are concerned with knowledge compilation and subsequent tuning, the instructional goals for mastery of certain skills may vary from an *intermediate* to a *high* performance level.

Worked-out examples

In the declarative stage, students usually study instructional materials or listen to lectures to encode declarative information. After these activities they start practicing, which often involves program generation, program modification, or program amplification. Anderson *et al.* (1984) reported that students in this stage of knowledge compilation make a highly selective use of instructional materials. In particular, they use concrete examples of problem solutions – related to the problem at hand – that have the form of concrete computer programs. These worked-out examples function as *analogies*, which students use as blue-prints or concrete schemata to map their new solutions. Thus, analogy is used to bridge the gap between the current declarative knowledge and the desired programming behavior. After students have gained more experience, their need for worked-out examples disappears, as a result of knowledge compilation.

The key to the use of analogy is interpreting information by general productions. Whereas in elaborative processing productions interpret schemata and create elaborations by mapping the schemata onto the instructional material, in the use of analogy productions interpret worked-out examples and create new solutions by mapping the examples onto existing declarative knowledge. Thus, students transform a worked-out example that is the solution for one problem into a new program that is the solution for another problem by interpreting the example and mapping it onto existing knowledge about programming. Obviously, analogy is a powerful tool in guiding programming behavior but it is never an automatic map-

ping of the example onto the new solution: students always need newly acquired knowledge about programming to reach a correct solution.

An instructional implication of ACT* is that students have to induce generalizations and discriminations from carefully selected examples because these are productions created by the automatic learning mechanisms of generalization and discrimination. However, in the PUPS successor of ACT*, generalizations and discriminations are seen as declarative knowledge structures that are produced by general problem solving productions. This leads to the additional implication that one should explicitly tell the students what the critical features in an example are (Anderson *et al.*, 1986). Thus, whereas the presentation of worked-out examples is important in its own right, the examples should be *annotated* with information about what they are supposed to illustrate.

Annotated examples bear resemblance to programming plans: they both offer templates of code instead of unorganized factual information and they both stress the critical features in this template. But, whereas programming plans primarily serve to present new information concerning a template of programming code and its relationship with specific programming problems, annotated, worked-out examples serve as an analogy to support knowledge compilation. In fact, we think that it is desirable to further annotate worked-out examples by explicitly referring to the programming plans they use.

Based on the function of worked-out examples as analogies that both guide programming behavior and support knowledge compilation, we present our first tactic for procedural instruction:

Tactic of worked-out examples

GOAL(S):

- intermediate performance level in program generation

CIRCUMSTANCE(S):

- students are novices in computer programming and the necessary declarative knowledge is already present

METHOD:

- present concrete, annotated, worked-out examples in the form of concrete programs for well-described programming problems that are related to the problems at hand

Basic cognitive skills

In the declarative stage, knowledge must continually be represented in working memory to be interpreted by general productions. The content of working memory rapidly changes because the interpretative production steps usually are small. The resulting high processing load has major costs in terms of speed as well as errors. Knowledge compilation decreases the load on working memory, both because

declarative information is built up into productions (proceduralization) so that it needs no longer be represented in working memory, and because production steps become larger (composition) so that the content of working memory is changing less frequently.

Basically, ACT* is a theory of learning by doing because practice is seen as a necessary condition for knowledge compilation. Practice may rapidly produce task-specific productions that result in a decrease of processing load. The same processes work for the compilation of productions that concern either basic skills or higher skills involved in programming. For instance, practice may build productions that help students to proceed in the programming environment, to apply syntactic rules, to couple particular programming problems to templates of language code, or to reformulate specific programming problems.

The idea is that students may have difficulties with higher skills involved in programming because the necessary basic skills have not been sufficiently practiced (*e.g.*, Resnick and Ford, 1981). By building up task-specific productions for basic cognitive skills, processing efficiency is increased so that the cognitive system is able to simultaneously perform another, higher-order task which does make demands on working memory. Although it is clearly impossible, given the available time, to train expert programmers in introductory programming courses, even a modest performance level in programming requires that several basic cognitive skills are learned up to the procedural stage so that the attention can be paid to the more complicated aspects of the total task. This leads us to the second tactic for procedural instruction:

Basic skills tactic

GOAL(S):

- high performance level in basic cognitive skills such as those involved in proceeding in the programming environment and applying syntactic rules

CIRCUMSTANCE(S):

- students are novices in computer programming and the necessary declarative knowledge is already present

METHOD:

- offer extensive practice in those basic skills

Task variation

It takes at least 100 hours to achieve only a very modest facility in programming skill (Anderson, 1982) and, in addition to formal training, several years of practical experience to become an expert programmer. Even after extensive training and practice, programmers with the same background generating a program for the same problem show large differences in performance; in addition, different problem specifications of the same difficulty lead to large differences for one programmer

(Barfield, Lebold, Salvendy and Shodja, 1983; Sackman, Erickson and Grant, 1968). The procedural knowledge base of an expert programmer is estimated to consist of ten to hundreds of thousands of highly task-specific productions (*e.g.*, Brooks, 1977), which all have to be acquired through the interpretative use of declarative knowledge. This explains why learning to program is a lengthy process and why there are considerable performance differences between and within programmers.

An instructional implication is that there must be enough task variation in practice to develop a broad procedural knowledge base, which underlies flexibility in programming behavior on a high performance level. This implication is particularly important for the training of professional programmers; in introductory programming courses at high school level there is neither occasion for extensive practice of all skills involved in programming nor for much task variation within practice. However, some variation in elementary programming may be offered by (a) the assignment of different tasks, such as using the editor, comprehending programs, designing algorithms, generating programs, debugging programs and so forth, and (b) the presentation of a broad range of both programming problems that have different underlying solutions in program generation and programs that are the solutions for different programming problems in program comprehension.

Offering task variation explicitly aims at the compilation – and subsequent tuning – of a broad procedural knowledge base. We think that in elementary programming it is at least equally important to support the development of schema-like, declarative knowledge structures. Interpreting such declarative structures by general productions also offers flexibility in programming behavior; although this has certain disadvantages in terms of speed and errors, it certainly is a realistic instructional goal to strive for in elementary programming. For this reason and in accordance with our discussion of worked-out examples, it is not only important to offer students some variation in problems and programs but it is also important to tell them what the critical features in these different problems and programs are. Based upon the function of task variation for the development of procedural knowledge, we present our last tactic for procedural instruction:

Task variation tactic

GOAL(S):

- intermediate/high performance level and flexibility in program comprehension and generation

CIRCUMSTANCE(S):

- students are novices/intermediates and the necessary declarative knowledge is already present

METHOD:

- offer variation in the different skills involved in computer programming and present a wide range of programming problems and programs

3.4 Processing load, strategies and tactics

Novice programmers make errors that may either be contributed to their misconceptions or to processing overload (Gilmore, 1986). Errors that only occur when tasks become more complex – where the meaning of "complex" changes as students gain more experience – may be explained by a processing overload model; errors that are independent of task complexity may be contributed to misconceptions. According to Anderson and Jeffries (1985), processing overload places the more serious constraints on both problem solving performance and learning in computer programming.

Students' errors and slowness resulting from processing overload are clearly shown in their behavior when they generate programs. Whereas experts show a systematic top-down, breadth-first expansion of the solution, novices show less structured and more opportunistic behavior. Such behavior may be explained either by a model of opportunistic planning (*e.g.*, Hayes-Roth and Hayes-Roth, 1979) or by a goal-directed system, such as GRAPES, that is subject to serious failures of working memory resulting from processing overload. That is, novices who show opportunistic behavior simply forget their goals as a result of processing overload and after losing their goal structures from working memory, they analyze the current state and construct some partial solution to it.

Groups of instructional strategies for designing programming courses may be distinguished by their approach to controlling processing load. The Spiral approach controls processing load by offering simple coding problems in the beginning of the course and more complex design problems only later in the course; the Reading approach controls processing load by varying the difficulty of the students' task from reading, through modification and amplification, to coding and designing complete programs. In the Expert approach, the complexity of the presented design problems gradually increases during the course but the problems involve algorithm design from the outset. Thus, they are relatively complex and may cause processing overload; this is one of the reasons that students are urged to adopt top-down programming techniques. Working according to a top-down model should enable students to assign working memory capacity to, successively, high level goals that include reformulation of the problem and design of the algorithm, low level goals that include finding solutions for subproblems, and achievable actions that include coding program statements.

However, top-down design techniques may minimize processing load for expert programmers but not for novices. Strictly speaking, top-down programming is possible if students have at each step available an appropriate set of productions as well as the necessary declarative knowledge. This only occurs if both the problem is of a familiar type and the student has experience with the programming language. When top-down programming is possible, it will minimize processing load; however, when it is not possible – as will often be the case for

novices – it *cannot* prevent processing overload. Consequently, top-down programming in introductory programming courses may be desirable, but it is often not possible because the necessary knowledge is not available.

Summarizing, instructional strategies differ in great measure in the way they control processing load. We stated that the Expert approach possibly succeeds in this to a lesser degree than the other approaches. In addition, the way processing load is controlled determines the global structure of an instructional strategy. As we will see in the next section, this has serious consequences for the facility of adopting instructional tactics in those strategies.

4 An evaluation of instructional strategies

Before actually starting an evaluation of instructional strategies, we will discuss how such an evaluation can take place. Obviously, the six instructional tactics can be used to evaluate the design of concrete introductory programming courses. The tactics constitute available knowledge from cognitive theory and empirical research directed towards learning to program. So, an effective programming course should incorporate those tactics.

Adopting a particular instructional strategy for the design of a programming course has consequences for the instructional tactics to be applied. Some tactics can be effectively applied in courses designed according to the strategy; others are less compatible with the strategy. If more strategies are available and only one is compatible with all tactics, this strategy is clearly superior. In this sense, tactics can be used to evaluate not only the design of actual courses but also the strategies underlying the designs. In this section, we make such a comparative evaluation for the three groups of instructional strategies that we found dominant in designing introductory programming courses.

An evaluation of strategies is only possible if they underlay the design of courses with the same goals. We base our evaluation of strategies on the assumption that the global goal of all courses is to take students as close as possible to the state of expert programmer. As a consequence, all courses are supposed to be directed towards program comprehension and generation paying attention to both design and coding aspects of programming. Furthermore, we assume that basic cognitive skills, such as those involved in proceeding in the programming environment and in applying syntactic rules of the language, should be practiced up to a high performance level; higher skills, such as those directly involved in program comprehension and generation, can only be learned up to an intermediate performance level because time limits prohibit the setting of higher learning outcomes. Thus, according to our evaluation, all three groups of instructional strategies should be compatible with all six tactics.

What does compatibility between strategies and tactics mean here? This question should be answered differently for declarative and procedural instruction. Declarative instruction takes place before students start practicing. Irrespective of the underlying strategy, it is possible to present all kinds of information to students, and thus all three elements of declarative instruction discussed, namely a computer model, programming plans and a design diagram. However, what is of main importance is that the declarative instruction should well prepare for subsequent practice. In other words, the central question to be answered is: "To what degree does declarative instruction deliver knowledge that is actually used in practice, when the student is performing tasks that are typical of the courses designed according to the particular strategy?".

Compatibility of procedural tactics with an instructional strategy is of another nature. The Expert, Spiral and Reading approach mainly differ from one another in the way they control processing load. The danger of processing overload is especially present when students perform programming tasks; that is, when they receive procedural instruction. As to the evaluation of strategies, relevant questions to ask include: "What help is given for problem solving?", "What kinds of problems and tasks are assigned?" and "How is task variation accomplished?" Thus, compatibility between a strategy and a procedural tactic stands for self-evidence of incorporation of this tactic in a course designed according to the strategy. That is, a course should contain worked-out examples, extensive practice in basic skills and some task variation.

4.1 Evaluation of the Expert approach

The Expert approach emphasizes systematic, top-down algorithm and program design. Program generation is presented as a primary student activity and design aspects are stressed by the presentation of non-trivial design problems from the outset of the course.

Declarative instruction

Knowledge of a computer model is especially useful when students must comprehend or generate lines of program code, such as for instance an assignment statement or a loop structure in Pascal. The Expert approach, however, heavily emphasizes design aspects in program generation; in most presented tasks the design of an algorithm forms the kernel of the problem. Consequently, we do not expect much result of the presentation of a model and we conclude that there is a low compatibility between the Computer model tactic and the Expert approach. This low compatibility may explain why this tactic is only sporadically applied in the Expert approach.

The presentation of programming plans is very suitable to an approach that is directed toward teaching expert behavior. Without doubt, students will often use knowledge of programming plans when they solve the design problems that are typical of this approach. We expect a high compatibility between the Programming-plans tactic and the Expert approach. The fact that programming plans begin to appear more and more in courses and textbooks categorized under the Expert approach gives support to this expectation.

Whereas novice programmers usually have great difficulties in coordinating their activities, experts systematically use a top-down model of successive refinement to coordinate their activities in algorithm and program design. The Expert approach tries to model expert behavior; this can be done by explicitly presenting a design diagram. Assuming that students believe in its effectiveness, they can profitably use it in solving the design problems that characterize this approach. Thus, we assume a high compatibility between the Design diagram tactic and the Expert approach.

Procedural instruction

Most textbooks labelled under the Expert approach contain sufficient worked-out examples. Clearly, the importance of such instructional elements is widely realized by those who adhere to the Expert approach. However, these examples generally function as declarative instruction; they are presented in isolation from the tasks assigned as practice. While trying to solve problems, students have to search for examples that fit in with their solution and they must turn back leaves, looking for examples analogous with 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. Therefore, we expect a moderate compatibility between the Tactic of worked-out examples and the Expert approach.

Whereas a generally accepted principle within the Expert approach is that students should get enough opportunity to practice, design activities are heavily emphasized and little attention is given to practice of basic skills. For this reason, we think that there is only a moderate compatibility between the Basic skills tactic and the Expert approach. In addition, neglecting the importance of practicing basic skills may further hamper an effective control of processing load.

Task variation is difficult to realize in the Expert approach as design aspects in program generation are emphasized at the cost of coding aspects, program comprehension and basic skills; furthermore, variation in offered problems and programs is low. The presentation of non-trivial design problems from the outset of the course takes a great amount of the available instructional time; usually, one programming problem engages the students' attention during one or two lesson periods. For this reason, the total number of problems that is presented to the

students remains small. We conclude that there is a low compatibility between the Task variation tactic and the Expert approach.

In short, we assume a high compatibility of the Expert approach with two tactics (programming plans, design diagram), a moderate compatibility with two tactics (worked-out examples, basic skills), and a low compatibility with two tactics (computer model, task variation). In addition, we like to recall that the control of processing load is probably ineffective in the Expert approach.

4.2 Evaluation of the Spiral approach

The Spiral approach is characterized by stepwise incremental learning. Program generation is presented as a primary student activity. Initially, coding activities are emphasized; the assigned tasks are mainly trivial coding problems requiring only syntactic and low level semantic knowledge. Gradually, when the students gain more experience, the assigned tasks become more complex and attention is directed to design aspects of programming.

Declarative instruction

Knowledge of a computer model is especially helpful in comprehending and generating lines of program code. Tasks of this kind are, at least during the main part of the course, emphasized in the Spiral approach. Furthermore, it should be recalled that research demonstrating the effectiveness of computer models was restricted to courses that we assign to the Spiral Approach. Therefore, we expect a high compatibility between the Computer model tactic and the Spiral approach.

To our knowledge, programming plans have never been presented in courses designed according to the Spiral approach. However, we expect knowledge of programming plans to be extremely useful for tasks that stress design aspects as well as for tasks stressing coding aspects in programming. Thus, we conclude that there is a high compatibility between the Programming-plans tactic and the Spiral approach.

Knowledge of a design diagram is especially useful in solving design problems. In the Spiral approach, such problems appear only late in the course so that the usefulness of knowledge of a design diagram is limited. In addition, this diagram can only be sensibly presented towards the end of the course, when genuine design problems are assigned; in the meantime, students may have acquired bad programming habits that are difficult to unlearn. As these habits may violate the ideas underlying systematic design, we expect a low compatibility between the Design diagram tactic and the Spiral approach.

Procedural instruction

The presentation of worked-out examples is well possible in courses designed according to the Spiral approach. In fact, example programs can be found in most

textbooks that we assign to this approach. The examples are usually solutions for trivial coding problems in the form of rather simple programs. As in the Expert approach, these examples often function as declarative instruction, but because of their simple nature we expect students to have fewer difficulties in finding examples that match the solutions of problems they are trying to solve. Thus, there is a moderate compatibility between the Tactic of worked-out examples and the Spiral approach.

Stepwise incremental learning in the Spiral approach warrants application of the Basic skills tactic because each separate step is finished with practice and there is an initial emphasis on coding activities. For instance, students are forced to extensive practice in using the programming environment and applying syntactic rules of the language. Clearly, a high compatibility exists between the Basic skills tactic and the Spiral approach.

Students' activities in the Spiral approach are especially directed towards the generation of programs at the cost of program comprehension. Although many tasks or problems may be assigned, they show a strong tendency, during the main part of the course, to solve trivial coding problems paying little attention to design aspects. Therefore, we conclude that there is a low compatibility between the Task variation tactic and the Spiral approach.

Summarizing, we assume a high compatibility of the Spiral approach with three tactics (computer model, programming plans, basic skills), a moderate compatibility with one tactic (worked-out examples), and a low compatibility with two tactics (design diagram, task variation).

4.3 Evaluation of the Reading approach

The Reading approach is characterized by its emphasis on program comprehension, modification and amplification. From the beginning of the course, students are confronted with program reading assignments in the form of non-trivial design problems in combination with their complete or partial solutions. The assigned tasks gradually become more complex during the course, changing from using and analyzing programs, through modifying and extending programs, to designing and coding programs.

Declarative instruction

Knowledge of a computer model helps students to comprehend lines of program code. In courses that have been designed according to the Reading approach most assigned tasks require the comprehension of code; especially in the beginning of the course, students are required to comprehend separate program lines while tracing programs. So, we expect a high compatibility between the Computer model tactic and the Reading approach.

Programming plans can easily be presented in the Reading approach. In fact, the solutions of the presented problems may be further annotated by explicitly referring to the plans they use. Knowledge of programming plans is not only useful for program comprehension – as for instance required in program modification and amplification – but also for design as well as coding activities in program generation. Thus, we assume a perfect compatibility between the Programming-plans tactic and the Reading approach.

Knowledge of a design diagram is particularly useful in solving non-trivial design problems. According to the Reading approach, such genuine design problems are assigned only late in the course. However, in contrast to the Spiral approach, a design diagram may be sensibly presented in an early phase of the course: In combination with the presented well-structured programs, the design diagram illustrates the usefulness of top-down design techniques and systematic planning. Students actually see that programming according to a design diagram helps to understand algorithms and perform other tasks that are typical of the Reading approach. We conclude that there certainly is a moderate compatibility between the Design diagram tactic and the Reading approach.

Procedural instruction

Presenting worked-out examples automatically takes place in courses designed according to the Reading approach. Unlike worked-out examples in the Expert and Spiral approaches, there is a direct bond between examples and practice in the Reading approach. Furthermore, these examples concern *working* programs that can be well-documented and annotated with information about what they are supposed to illustrate. Obviously, the compatibility between the Tactic of worked-out examples and the Reading approach is very high.

Practice in basic skills is very well possible within the Reading approach. Students immediately start handling the computer and using the programming environment when they run given programs; soon after, they have to apply syntactic rules when tasks are assigned to them in which programs have to be modified or amplified. Thus, students start with intensive practice of basic skills on a modest scale. We expect a high compatibility between the Basic skills tactic and the Reading approach.

Task variation is easily accomplished as all kinds of tasks, such as using the editor, applying syntax, interpreting programs and generating programs appear in the course. Whereas the complexity of the presented design problems in the Expert approach is high and only few problems can be presented, the difficulty of assigned tasks in the Reading approach is better controllable so that the students can be confronted with a greater amount of problems. Furthermore, these problems considerably vary in design and coding activities they require, unlike the often trivial coding problems in the Spiral approach. Finally, the presentation of a wide range of example programs is inherent in the Reading approach. For these

reasons, we conclude that there is a high compatibility between the Task variation tactic and the Reading approach.

In short, we assume a high compatibility of the Reading approach with five tactics (computer model, programming plans, worked-out examples, basic skills, task variation) and a moderate compatibility with only one tactic (design diagram).

4.4 A final comparison of strategies

The results of the separate evaluations of groups of instructional strategies are displayed in Table 1. Based on these results, a comparative evaluation of the Expert, Spiral and Reading approach is possible.

Table 1. Compatibility between Instructional Strategies and Instructional Tactics

	Strategies		
Tactics	Expert Approach	Spiral Approach	Reading Approach
		Declarative Instruction	
Computer Models	-	+	+
Programming Plans	+	+	+
Design Diagrams	+	-	-/+
		Procedural Instruction	
Worked-out Examples	-/+	-/+	+
Basic Skills	-/+	+	+
Task Variation	-	-	+

Note. - Low, -/+ Moderate, + High

According to Table 1, the Reading approach scores higher than or equal to the other groups of instructional strategies on five out of six instructional tactics. If all tactics get equal weight, the Reading approach is the best strategy to follow in the instructional design of introductory programming courses. The superiority of the Reading approach to the Spiral approach is, given our evaluation, beyond dispute. The Expert approach would only be able to compete with the Reading approach if the Design diagram tactic was given very much weight; as we feel that – as an extra benefit – the Reading approach is superior in its control of processing load, this approach is proclaimed to be the best strategy according to our evaluation.

A direct comparison of the Expert approach with the Spiral approach is less easy. The Expert approach should be preferred if the Design diagram is considered

very important; on the other hand, the Spiral approach should be preferred if both the Computer model tactic and the Basic skills tactic are considered as more important than the other tactics. As a speculation, we think that courses designed according to the Spiral approach will be slightly more successful, in part thanks to their better control of processing load.

5 Discussion and research implications

We offered a theoretical framework for the instructional design of introductory programming courses. In this framework, the distinction between instructional strategies and instructional tactics on the one hand, and between declarative instruction and procedural instruction on the other hand, was emphasized. Instructional strategies were defined as general design plans that mainly differ in their control of students' processing load but that all pursue the same global instructional goals. We described the Expert, Spiral and Reading approach as three groups of closely related strategies. In order, they emphasize top-down design techniques, incremental learning, and program modification and amplification.

Against the background of ACT*, we stressed the distinction between declarative instruction and procedural instruction. Declarative instruction involves the initial presentation of new information; procedural instruction involves the design of practice. We presented six instructional tactics that could partially be supported by relevant research. Tactics were defined as specific plans of action that prescribe methods to reach desired learning outcomes under given circumstances. Tactics for declarative instruction include concrete computer models, programming plans and design diagrams; tactics for procedural instruction include worked-out examples, practice of basic cognitive skills and task variation. Tactics may be used both to design new courses and to evaluate existing strategies. In our evaluation of the three groups of instructional strategies, the Reading approach was found to be superior to the Expert and the Spiral approach.

The framework presented is an attempt to organize available knowledge for the design of introductory programming courses. However, it is not yet completed as several aspects that we feel to be important for instructional design have been omitted. First, these include management aspects of instruction; that is, we did not present instructional tactics that depend on class size. Second, several areas of research may lead to valuable instructional tactics that can eventually be included in our framework. For instance, declarative instruction may be further improved by applying tactics that take students' misconceptions into account; procedural instruction may be further improved by tactics that offer methods for individualized tutoring of problem solving processes involved in programming.

Furthermore, the role of several "intermediate" products that may be used in instruction, such as flow-charts, pseudo-programming languages and structure-

diagrams, has not been discussed. Programmers may profit from the use of such intermediates because they provide a clear separation of the design and coding activities involved in programming. The idea is that design activities should result in a flow-chart, pseudo-program, or structure-diagram which may subsequently be translated into the programming code at hand. Whereas, to our knowledge, a positive effect of using intermediates on learning outcomes has never been clearly demonstrated in elementary programming, it may obviously have important consequences for instructional design.

Our analysis of instructional strategies and tactics is subject to some further limitations. First, the distinction between three groups of instructional strategies is based upon an investigation of textbooks, curricula and point-of-view papers in which clear differences appeared in the way students' processing load was controlled. However, most instructional materials and courses really used a mixture that was only leaning towards one of the approaches and, whereas there were many instructional materials that could be assigned to the Expert or the Spiral approach, there were only few materials available that could be classified under the Reading approach. Second, instructional tactics were not exclusively based on empirical research; some of them were formulated against the theoretical background of ACT* and should be seen as hypotheses that have yet to be confirmed. Clearly, the development of new insights into current psychological theory will have direct consequences for our framework.

Another difficulty is that the steps that were taken from the psychological theory to the formulation of instructional tactics were not – and in our opinion, cannot be – straightforward. A psychological theory usually does not provide enough information to give a precise description of the circumstances under which a tactic is valuable and, on the other hand, the methods to reach desired learning outcomes are only implicitly present. For instance, it is not clear if the presentation of a computer model is valuable in the teaching of a functional programming language (*e.g.*, LISP) instead of an imperative language (*e.g.*, BASIC, Pascal); moreover, if this should be the case, it is still not clear what form the new computer model should have.

In our opinion, the tracing, formulation and confirmation of instructional tactics should be a first concern for research on instructional design for introductory programming courses. This research should focus upon (a) an extension of the set of tactics, (b) a refinement of goals, circumstances and methods in tactics, (c) an assignment of weights to tactics and (d) an assessment of interactions between tactics. By coupling the tactics to a common theoretical background it will eventually become possible to build a coherent, comprehensive framework for instructional design in elementary programming.

A second, strongly related concern should be a comparison of instructional strategies that apply different sets of tactics. Our evaluation of the Expert, Spiral and Reading approach was based upon our weak assumptions regarding the appli-

cability of tactics; however, a strict arrangement was not possible because we lacked more precise information about those tactics. By making careful comparisons of learning outcomes in courses designed according to strategies that apply different tactics, we think it will be possible both to gather more information about the tactics applied and to track down global guidelines for instructional design.

Future research on instructional design for elementary computer programming should also take into account other important features of strategies, such as management aspects, students' and teachers' motivation, and possibilities of individualized tutoring. The significance of such research is clear. Whereas the teaching of computer programming in high schools is already very common now, and the research concerning the psychological processes involved in programming is quickly developing, there are still few guidelines that teachers and others in the educational field can use for their design of introductory programming courses.

Notes

The authors wish to express their gratitude to Sanne Dijkstra, Otto Jelsma and Georg Rakers for their helpful comments on a draft of this article. Correspondence concerning this article should be addressed to Jeroen J. G. van Merrienboer.

References

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory and Cognition, 9,* 422-433.

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

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

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

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

Anderson, J. R., Greeno, J. G., Kline, P. J. and Neves, D. M. (1981). Acquisition of problem solving skill. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 191-230). Hillsdale, NJ.: Erlbaum Associates.

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

Anderson, J. R., Kline, P. J. and Beasly, C. M. (1980). Complex learning processes. In R. E. Snow, P. A. Federico and W. E. Montague (Eds.), *Aptitude, learning and instruction: Vol. 2.* Hillsdale, NJ.: Erlbaum Associates.

Anderson, J. R. and Reiser, B. J. (1985). The LISP tutor. *Byte, 10(4),* 159-175.

Anderson, J. R. and Skwarecki, E. (1986). The automated tutoring of introductory computer programming. *Communications of the ACM, 29,* 842-849.

Atherton, R. (1981). Requirements for a general purpose high-level programming language for schools. *Computer Education, 38,* 14-16.

Atherton, R. (1982). BASIC damages the brain. *Computer Education, 40,* 14-17.

Atwood, M. E., Turner, A. A., Ramsey, H. R., Hooper, J. N. and Sidorsky, R. C. (1979). *An exploratory study of the cognitive structures underlying the comprehension of software design problems* (Tech. Rep. No. 392). Alexandria, VA.: US Army Research Institute for the Behavioral and Social Sciences.

Ausubel, D. P. (1968). *Educational psychology: A cognitive approach.* New York: Holt, Rinehart & Winston.

Baird, W. E. (1982). *Problem solving, teamwork and structured programming - in high school?* In J. Smith and G. Schuster Moon (Eds.), Proceedings of the NECC (pp. 331-334). Columbia: University of Missouri.

Balzert, H. and Hille, H. (1980). A standardized curriculum in informatics for adult education. *Computers and Education, 4,* 189-198.

Barfield, W. (1986). Expert-novice differences for software: Implications for problem-solving and knowledge acquisition. *Behavior and Information Technology, 5,* 15-29.

Barfield, W., LeBold, W. K., Salvendy, G. and Shodja, S. (1983). *Cognitive factors related to computer programming and software productivity.* In Proceedings of the Human Factors Society, 27th Annual Meeting (pp. 647-651). Norfolk, VA.: Human Factors Society.

Baron, J., Szymanski, B., Lock, E. and Prywes, N. (1985). An argument for non-procedural languages. In R. Jernigan, B. W. Hamill and D. M. Weinstraub (Eds.), *The role of language in problem solving* (pp. 127-144). Amsterdam, North Holland: Elsevier Science Publishing.

Bayman, P. and Mayer, R. E. (1983). A diagnosis of beginning programmer's misconceptions of BASIC programming statements. *Communications of the ACM, 26,* 677-679.

Bonar, J. and Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction, 1,* 133-161.

Bradley, C. A. (1985). The relationship between students' information processing style and Logo programming. *Journal of Educational Computing Research, 1,* 427-433.

Brooke, J. B. and Duncan, K. D. (1980). Experimental studies of flowchart use at different stages of program debugging. *Ergonomics, 23,* 1057-1091.

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

Clements, D. H. (1986a). Effects of Logo and CAI environments on cognition and creativity. *Journal of Educational Psychology, 78,* 309-318.

Clements, D. H. (1986b). Logo and Cognition: A theoretical foundation. *Computers in Human Behavior, 2,* 95-110.

Clements, D. H. (1987). Longitudinal study of the effects of Logo programming on cognitive abilities and achievement. *Journal of Educational Computing Research, 3,* 73-94.

Clements, D. H. and Gullo, D. F. (1984). Effects of computer programming on young children's cognition. *Journal of Educational Psychology, 76,* 1051-1058.

Cooper, D. and Clancy, M. (1982). *Oh! Pascal!.* New York: W. W. Norton.

Curtis, B. (1983, March). A review of human factors research on programming languages and specifications. *Monitor,* pp. 24-30.

Dahl, O. J., Dijkstra, E. W. and Hoare, C. A. R. (1972). *Structured programming.* London: Academic Press.

Dalbey, J. and Linn, M. C. (1986). Cognitive consequences of programming: Augmentations to basic instruction. *Journal of Educational Computing Research, 2,* 75-93.

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

Dale, N. and Orschalick, D. (1983). *Introduction to Pascal and structured design.* New York: D. C. Heath and Co.

Deimel, L. E. and 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.

Dershem, H. L. (1980). *Computer problem solving (Modules and monographs in undergraduate mathematics and its applications project).* Newton, MA.: Educational Development Center.

Dijkstra, E. W. (1968). GOTO statement considered harmful. *Communications of the ACM, 11,* 147-148.

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

DuBoulay, B., O'Shea, T. and Monk, J. (1981). The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Man-Machine Studies, 14,* 237-249.

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

Ehrlich, K., Soloway, E. and Abbott, V. (1982). *Transfer effects from programming to algebra word problems: A preliminary study* (Tech. Rep. No. 257). New Haven, CT.: Yale University, Dept. of Computer Science.

Fitter, M. J. and Green, T. R. G. (1979). When do diagrams make good computer languages? *International Journal of Man-Machine Studies, 11,* 235-261.

Gall, M. (1984). Synthesis of research on teachers' questioning. *Educational Leadership, 42 (3),* 40-47.

Gilmore, D. J. (1986). Structural visibility and program comprehension. In M. D. Harrison and A. F. Monk (Eds.), *People and computers: Designing for usability* (pp. 527-545). Cambridge: Cambridge University Press.

Green, T. R. G. (1982). Pictures of programs and other processes, or how to do things with lines. *Behaviour and Information Technology, 1,* 3-36.

Green, T. R. G. (1983). Learning big and little programming languages. In A. C. Wilkinson (Ed.), *Classroom computers and cognitive science* (pp. 71-93). New York: Academic Press.

Green, T. R. G., Sime, M. E. and Fitter, M. J. (1980). The problems the programmer faces. *Ergonomics, 23,* 893-907.

Hayes-Roth, B. and Hayes-Roth, F. (1979). A cognitive model of planning. *Cognitive Science, 3,* 275-310.

Hoc, J. M. (1981). Planning and direction of problem solving in structured programming: An empirical comparison between two methods. *International Journal of Man-Machine Studies, 15,* 363-383.

Jeffries, R., Turner, A. A., Polson, P. G. and 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.

Johnson, W. L. and Soloway, E. (1985a). PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering, 11,* 267-275.

Johnson, W. L. and Soloway, E. (1985b). PROUST: An automatic debugger for Pascal programs. *Byte, 10(4),* 179-193.

Joni, S. A. and Soloway, E. (1986). But my program runs! Discourse rules for novice programmers. *Journal of Educational Computing Research, 2,* 95-125.

Larkin, J. H. (1979). Information processing models and science instruction. In J. Lochhead and J. Clement (Eds.), *Cognitive process instruction* (pp. 109-118). Philadelphia, PA.: The Franklin Institute Press.

Linn, M. C. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher, 14(5),* 14-29.

Mandinach, E. B. and Linn, M. C. (1986). The cognitive effects of computer learning environments. *Journal of Educational Computing Research, 2,* 411-427.

Mayer, R. E. (1975). Different problem solving competencies established in learning computer programming with and without meaningful models. *Journal of Educational Psychology, 67,* 725-734.

Mayer, R. E. (1976). Some conditions of meaningful learning for computer programming: Advanced organizers and subject control of frame order. *Journal of Educational Psychology, 68,* 143-150.

Mayer, R. E. (1979). A psychology of learning BASIC. *Communications of the ACM, 22,* 589-593.

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 and B. Hunter (Eds.), *Computer literacy* (pp. 129-159). New York: Academic Press.

Mayer, R. E. and Bromage, B. (1980). Different recall protocols for technical text due to advance organizers. *Journal of Educational Psychology, 72,* 209-225.

McKeithen, K. B., Reitman, J. S., Rueter, H. H. and Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology, 13,* 307-325.

Mettes, C. T. W., Pilot, A. and Roossink, H. J. (1981). Linking factual and procedural knowledge in solving science problems: A case study in a thermodynamics course. *Instructional Science, 10,* 333-361.

Neves, D. M. and Anderson, J. R. (1982). Knowledge compilation: mechanisms for the automatization of cognitive skills. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 57-84). Hillsdale, NJ.: Erlbaum Associates.

Newell, A. and Simon, H. A. (1972). *Human Problem Solving.* Englewood Cliffs, NJ.: Prentice-Hall.

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

Reigeluth, C. M. (1983). Instructional design: What is it and why is it? In C. M. Reigeluth (Ed.), *Instructional-design theory and models* (pp. 3-36). Hillsdale, NJ.: Erlbaum Associates.

Reiser, B. J. Anderson, J. R. and Farrell, R. G. (1985). *Dynamic student modelling in an intelligent tutor for LISP programming.* In Proceedings of IJCAI-85 (pp. 8-14). Los Angeles: IJCAI.

Resnick, L. B. and Ford, W. W. (1981). *The psychology of mathematics for instruction.* Hillsdale, NJ.: Erlbaum Associates.

Sacerdoti, E. G. (1977). *A structure for plans and behavior.* New York: Elsevier Science Publishing.

Sackman, H., Erickson, W. J. and Grant, E. E. (1968). Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM, 11,* 3-14.

Samurçay, R. (1985). Learning programming: An analysis of looping strategies used by beginning students. *Journal for the Learning of Mathematics, 5,* 37-43.

Schulz-Zander, R. (1981). Ein didaktischer Ansatz für den Informatikunterricht [A didactic proposal for computer science education]. *Log In, 1,* 24-27.

Schulz-Zander, R. (1986). *Auswirkungen von Programmiersprachen auf das Problem löseverhalten von Schülern* [Effects of programming languages on students' problem-solving behavior]. Kiel: Institut für die Pädagogik der Naturwissenschaften an der Universität Kiel.

Sheppard, S. B., Kruesi, E. and Curtis, B. (1981). *The effects of symbology and spatial arrangement on the comprehension of software specifications.* In Proceedings of the Fifth International Conference on Software Engineering (pp. 207-214). Silver Spring, MD.: IEEE computer society.

Shneiderman, B. (1976). Exploratory experiments in programmer behavior. *International Journal of Computer and Information Sciences, 5,* 123-143.

Shneiderman, B. (1977a). Teaching programming: A spiral approach to syntax and semantics. *Computers and Education, 1,* 193-197.

Shneiderman, B. (1977b). Measuring computer program quality and comprehension. *International Journal of Man-Machine Studies, 9,* 46-59.

Shneiderman, B. and Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences, 8,* 219-238.

Shneiderman, B., Mayer, R., McKay, D. and Heller, P. (1977). Experimental investigations of the utility of detailed flow-charts in programming. *Communications of the ACM, 20,* 373-381.

Sime, M. E., Arblaster, A. T. and Green, T. R. G. (1977). Structuring the programmer's task. *Journal of Occupational Psychology, 50,* 205-216.

Simon, H. A. (1980). Problem solving and education. In D. T. Tuma and F. Reif (Eds.), *Problem solving and education: Issues in teaching and research.* Hillsdale, NJ.: Erlbaum Associates.

Sleeman, D., Putnam, R. T., Baxter, J. and Kuspa, L. (1986). Pascal and high school students: A study of errors. *Journal of Educational Computing Research, 2,* 5-23.

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.

Soloway, E., Bonar, J. and Ehrlich, K. (1982). *Cognitive strategies and looping constructs: An empirical study* (Tech. Rep. No. 242). New Haven: Yale University, Dept. of Computer Science.

Soloway, E., Ehrlich, K., Bonar, J. and Greenspan, J. (1982). What do novices know about programming? In A. Badre and B. Shneiderman (Eds.), *Directions in human-computer interactions* (pp. 27-54). Norwood, NJ.: Ablex Publishing Corp.

Soloway, E., Lochhead, J. and Clement, J. (1982). Does computer programming enhance problem solving ability? Some positive evidence on algebra word problems. In R. Seidel, R. Anderson and B. Hunter (Eds.), *Computer literacy* (pp. 171-185). New York: Academic Press.

Spohrer, J. C., Soloway, E. and Pope, E. (1985). A goal/plan analysis of buggy Pascal programs. *Human-Computer Interaction, 1,* 163-207.

Webb, N. M. (1983). Predicting learning from student interaction: Defining the interaction variables. *Educational Psychologist, 18,* 33-41.

Webb. N. M. (1984). Microcomputer learning in small groups: cognitive requirements and group processes. *Journal of Educational Psychology, 76,* 1076-1089.

Webb. N. M., Ender, P. and Lewis, S. (1986). Problem solving strategies and group processes in small groups learning computer programming. *American Educational Research Journal, 32,* 243-261.

Weiser, M. and Shertz, J. (1983). Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies, 19,* 391-398.

Wirth, N. (1974). On the composition of well-structured programs. *Computing Surveys, 6,* 247-259.

Woodhouse, D. (1983). Introductory courses in computing: Aims and languages. *Computer Education, 7(2),* 79-89.

Yourdon, E. (1975). *Techniques of program structure and design.* Englewood Cliffs, NJ.: Prentice-Hall.