Computer Science Education
2003, Vol. 13, No. 2, pp. 137–172

0899-3408/03/1302-137$16.00
© Swets & Zeitlinger

Learning and Teaching Programming: A Review and Discussion

Anthony Robins, Janet Rountree, and Nathan Rountree
Computer Science, University of Otago, Dunedin, New Zealand

ABSTRACT

In this paper we review the literature relating to the psychological/educational study of programming. We identify general trends comparing novice and expert programmers, programming knowledge and strategies, program generation and comprehension, and object-oriented versus procedural programming. (We do not cover research relating specifically to other programming styles.) The main focus of the review is on novice programming and topics relating to novice teaching and learning. Various problems experienced by novices are identified, including issues relating to basic program design, to algorithmic complexity in certain language features, to the "fragility" of novice knowledge, and so on. We summarise this material and suggest some practical implications for teachers. We suggest that a key issue that emerges is the distinction between effective and ineffective novices. What characterises effective novices? Is it possible to identify the specific deficits of ineffective novices and help them to become effective learners of programming?

1. INTRODUCTION

Programming is a very useful skill and can be a rewarding career. In recent years the demand for programmers and student interest in programming have grown rapidly, and introductory programming courses have become increasingly popular. Learning to program is hard however. Novice programmers suffer from a wide range of difficulties and deficits. Programming courses are generally regarded as difficult, and often have the highest dropout rates. It is generally accepted that it takes about 10 years of experience to turn a novice into an expert programmer (Winslow, 1996).

Address correspondence to: Anthony Robins, Computer Science, University of Otago, Dunedin, New Zealand. Tel.: +64-3-479-8314. Fax: +64-3-479-8529. E-mail: anthony@cs.otago.ac.nz

What are the properties of expert programmers? What resources and processes are involved in creating or understanding a program? Since the 1970s there has been an interest in questions such as these, and in programming as a cognitive process. The literature relating to such topics is extensive, and was especially active in the late 1980s. A more recent trend is an emphasis on studies of object-oriented (OO) programming and its relationship to the traditional procedural approach.

Our interest in this broad field is focused by practical considerations. We teach a computer science introductory programming course, the kind often known as "CS1". Our goal is to provide the most effective learning environment and experience that we can for our students. Consequently, we are interested in understanding the processes of learning and teaching programming. Why is programming hard to learn? What are the cognitive requirements of the task? Are there successful and unsuccessful strategies for learners? What can we as teachers do to most effectively support novice programmers? The purpose of this paper is to review research relating to novices learning a first programming language. While some comparison of the procedural and OO programming styles is included, we do not cover research relating specifically to other programming styles (such as, e.g., functional or logic programming). We briefly explore issues relating to teaching, and a main goal of this review is to identify practical implications for teachers.

We begin with an overview (Section 2) of research into programming, identifying several significant trends. We then focus (Section 3) on novice programmers, exploring their capabilities and typical problems, their characteristic behaviours, and (Section 4) factors relating to course design and teaching. In the concluding discussion (Section 5) we summarise this material and suggest some practical implications for teachers. We also propose a framework which makes explicit some of the relationships between important topics explored in the literature, and highlight the significance of the distinction between effective and ineffective novices, in particular focusing on the strategies that they employ.

2. OVERVIEW

Studies of programming can be generally divided into two main categories, those with a software engineering perspective, and those with a psychological/ educational perspective. Software engineering based studies typically focus on experienced or professional programmers, often working in teams, and how to develop software projects effectively (see, e.g., Boehm, 1981; Brooks, 1995; Humphrey, 1999; Mills, 1993; Perlis, Sayward, & Shaw, 1981). Our interest is in novices and the initial development of individual programming skills. Although early learning should of course include the basics of good software engineering practice, learning to program is usually addressed from a psychological/educational perspective, and it is this literature that is the focus of the current review. Research has focused on topics such as program comprehension and generation, mental models, and the knowledge and skills required to program.

Two early books (Sackman, 1970; Weinberg, 1971) were significant in identifying programming as an area of psychological interest and stimulating research in the field. Sheil (1981) is an often cited early review, which very clearly sets out and discusses a range of methodological issues (see also Gilmore, 1990a). More recent books include Soloway and Spohrer (1989), which is explicitly focused on the novice programmer, and Hoc, Green, Samurçay, and Gillmore (1990). Drawing on these and other sources, we can identify the following general trends and topics.

2.1. Experts Versus Novices

It is generally agreed (Winslow, 1996) that it takes roughly 10 years to turn a novice into an expert programmer. There are several breakdowns of this continuum into stages, the most commonly cited being the five stages proposed by Dreyfus and Dreyfus (1986): novice, advanced beginner, competence, proficiency, and expert.

There are many studies of "expert" programmers (although some of these are based on graduate students who are probably only competent or proficient on the scale noted above). Studies of experts focus in particular on the sophisticated knowledge representations and problem solving strategies that they can employ (see, e.g., Détienne, 1990; Gilmore, 1990b; Visser & Hoc, 1990). In a survey of program understanding, von Mayrhauser and Vans (1994) summarise studies (in particular Guindon, 1990) noting that experts: have efficiently organised and specialised knowledge schemas; organise their knowledge according to functional characteristics such as the nature of the underlying algorithm (rather than superficial details such as language syntax); use both general problem solving strategies (such as divide-and-conquer) and specialised strategies; use specialised schemas and a top-down, breadth-first approach to efficiently decompose and understand programs; and are flexible in their approach to program comprehension and their willingness to abandon questionable hypotheses. Expert knowledge schemas also have associated testing and debugging strategies (Linn & Dalbey, 1989). Rist summarises many of the advantages of the expert programmer as follows:

> Expertise in programming should reduce variability in three ways: by defining the best way to approach the design task, by supplying a standard set of schemas to answer a question, and by constraining the choices about execution structure to the 'best' solutions. (Rist, 1995, p. 552)

Many of the characteristics of expert programmers are also characteristics of experts in general, as explored, for example, in other fields such as chess or mathematics. Experts are good at recognising, using and adapting patterns or schemas (and thus obviating the need for much explicit work or computation). They are faster, more accurate, and able to draw on a wide range of examples, sources of knowledge, and effective strategies.

By definition novices do not have many of the strengths of experts. Studies reviewed by Winslow (1996), for example, have concluded that novices are limited to surface and superficially organised knowledge, lack detailed mental models, fail to apply relevant knowledge, and approach programming "line by line" rather than using meaningful program "chunks" or structures. Studies collected in Soloway and Spohrer (1989) outline deficits in novices' understanding of various specific programming language constructs (such as variables, loops, arrays and recursion), note shortcomings in their planning and testing of code, explore more general issues relating to the use of program plans, show how prior knowledge can be a source of errors, and more. Novices are "very local and concrete in their comprehension of programs" (Wiedenbeck, Ramalingam, Sarasamma, & Corritore, 1999, p. 278). Since our main interest is in novices and the early stages of learning, we return to this topic in more detail in Section 3.

2.2. Knowledge Versus Strategies

Davies (1993) distinguishes between programming knowledge (of a declarative nature, e.g., being able to state how a "for" loop works) and programming strategies (the way knowledge is used and applied, e.g., using a "for" loop appropriately in a program).

Obviously, programming ability must rest on a foundation of knowledge about computers, a programming language or languages, programming tools and resources, and ideally theory and formal methods. Typical introductory programming textbooks devote most of their content to presenting knowledge about a particular language (elaborated with examples and exercises), and in our experience typical introductory programming courses are also "knowledge driven".

The majority of studies of programming have likewise focused on the content and structure of programming knowledge, see, for example, Brooks (1990) introducing a special issue of *International Journal of Man-Machine Studies* (Vol. 33, No. 3) devoted to this topic. One kind of representation is usually identified as central, namely a structured "chunk" of related knowledge, typically called a schema or plan.[1] For example, most programmers will have a schema for finding the average of the values stored in single dimensional array. Ormerod (1990) suggests that "A schema [. . .] consists of a set of propositions that are organised by their semantic content", and goes on to further distinguish plans, frames and scripts (see also Anderson, 2000).

As used in the literature, however, there is considerable flexibility and overlap in the interpretation of these terms. In an observation which captures both the central role of the schema/plan, and the vagueness of the definition and terminology, Rist notes:

> There is considerable evidence in the empirical study of programming that the plan is the basic cognitive chunk used in program design and understanding. Exactly what is meant by a program plan, however, has varied considerably between authors. (Rist, 1995, p. 514)

We will follow the usage adopted by each author when discussing the work of others, and ourselves use the term schema to refer to this general kind of representation.

As various authors, and in particular Davies (1993) have pointed out, however, knowledge is only part of the picture:

> Much of the literature concerned with understanding the nature of programming skill has focused explicitly on the declarative aspects of programmers' knowledge. This literature has sought to describe the nature of stereotypical programming knowledge structures and their organization. However, one major limitation of many of these knowledge-based theories

[1]"Plan" is often used to emphasize an "action oriented" rather that static interpretation. In other words, the term "schema" implies a "program as text" perspective, while the term "plan" implies a "programming as activity" perspective (Rogalski & Samurçay, 1990).

> is that they often fail to consider the way in which knowledge is used or applied. Another strand of literature is less well represented. This literature deals with the strategic aspects of programming skill and is directed towards an analysis of the strategies commonly employed by programmers in the generation and comprehension of programs. (Davies, 1993, p. 237)

For example, Widowski and Eyferth (1986) compared novice and expert programmers as they worked to understand programs which were either conventionally or unusually structured. Subjects could view the code one line at a time, and a "run" was defined as a sequential pass over a section of code. Experts tended to read conventional programs in long but infrequent runs (Widowski & Eyferth suggest they are employing a top-down conceptually driven strategy), and read unusual programs in short frequent runs (suggesting a bottom-up heuristic strategy). Novices tended to read both conventional and unconventional programs in the same way. The authors suggest that experts (even without relevant knowledge structures or plans) had more flexible strategies, and were better able to recognise and respond to novel situations.

Davies suggests that research should go beyond attempts to simply characterise the strategies employed by different kinds of programmer, and focus on why these strategies emerge, i.e. on "exploring the relationship between the development of structured representations of programming knowledge and the adoption of specific forms of strategy" (Davies, 1993, p. 238). In his subsequent review, Davies identifies as significant strategies relating to the general problem domain, the specific programming task, the programming language, and the "interaction media" (programming tools). We cover much of the material reviewed in the discussion of program comprehension and generation below.

2.3. Comprehension Versus Generation

Another significant distinction in the literature is between studies that explore program comprehension (where given the text of a program subjects have to demonstrate an understanding of how it works), and those that focus on program generation (where subjects have to create a part of or a whole program to perform some task/solve some problem).

Brooks (1977, 1983) was among the first to propose a model of program comprehension. The model is set in the context of various knowledge domains, such as the original *problem domain* (e.g., a "cargo-routing" problem), which is transformed and represented as values and structures in intermediate domains, and finally instantiated in the data structures and algorithms of a program in the *programming domain.*[2] Brooks suggests that programming involves formulating mappings from the problem domain (via intermediate domains) into the programming domain – a process which requires knowledge of both the structure of the domains and of the mappings between them.

Brooks describes program comprehension as a "top-down" and "hypothesis-driven" process. Brooks suggested that rather than studying programs line by line, subjects (assumed to be "expert" programmers) form hypotheses based on high-level domain and programming knowledge. These hypotheses are verified or falsified by searching the program for markers/ "beacons" which indicate the presence of specific structures or functions. Subjects may vary with respect to their domain knowledge, programming knowledge, and comprehension strategies. This fairly detailed model is able to account, Brooks claims, for observed variation in comprehension performance arising from such factors as the nature of the problem domain, variations in the program text, the effects of different comprehension tasks (e.g., modification vs. debugging) and the effects of individual differences. Davies (1993) reviews a range of studies that support Brooks' model. Other models of program comprehension are reviewed in von Mayrhauser and Vans (1994), including those proposed by Shneiderman and Mayer (1979), Soloway and Ehrlich (1984), Soloway et al. (1988), Letovsky (1986) and Pennington (1987a, b). Wiedenbeck et al. (1999) note that subjects' models of a program can be influenced by different task requirements, for example, modifying a program rather than simply answering questions about it.

Rist (1995) presents a comprehensive model of program generation (see also Rist, 1986a, 1986b, 1989, 1990). Knowledge is represented using nodes in internal memory (working, episodic, and semantic) or external memory (the program specification, notes, or the program itself). A node encodes an "action" that may range from a line of code, to chunks such as loops, to one or more routines of arbitrary size. Nodes are indexed using a tuple of the form ⟨role, goal, object⟩, for example, a read loop could be indexed as ⟨read, stream, –⟩. Nodes also have four "ports", *:use*, *:make*, *:obey* and *:control*, which allow them to be linked with respect to control flow and data flow. A program is built by starting

[2]The same domains are identified by Pennington (1987a, 1987b), based on the text comprehension model of van Dijk and Kintsch (1983).

with a search cue such as ⟨find, average, rainfall⟩, and retrieving from memory any matching node. Nodes can contain cues, so cues within the newly linked node are then expanded and linked in the same way. Linked systems of code that produce a specific output called plans, and common/useful plans are assumed to be stored by experts as schema-like knowledge structures.

Using these underlying knowledge representations a number of different design strategies can be implemented. A design strategy (in this specific definition) consists of a starting cue, a direction, a level, and a type of link to explore next (all design decisions are local, with no "supervising controller"). By varying these conditions within the model a range of different programmer strategies (in the general sense of word as discussed above) can be implemented, including typical novice and expert strategies. Experts can typically *retrieve* relevant plans from memory, and then generate code from the plan in linear order (from initialisation, to calculation, to output). Novices must typically *create* plans. This involves "focal expansion" – reasoning "backwards" from the goal to the focus (critical calculation/step/transaction), and then to the other necessary elements. Code generation begins with the central calculation, and builds the initialisations and other elements around it.

Rist notes that a realistic design process will involve "the interaction between a search [design] strategy and opportunistic design, plan creation and retrieval, working memory limitations, and the structure of the specification and the program" (Rist, 1995, p. 508). (Such practical considerations, especially the limited capacity of working memory, are also addressed in the "parsing-ginsarp" model of program generation (Green, Bellamy, & Parker, 1987).) Rist's model has been implemented in a program which generates Pascal programs from English descriptions.

Studies and models of comprehension are more numerous than studies and models of generation, possibly because comprehension is a more constrained task and subject's behaviour is therefore easier to interpret and describe. Clearly the topics are related, not least because during generation the development, debugging (and in the long term maintenance) of code necessarily involves reviewing and understanding it. Although we might therefore expect that these abilities will always be highly correlated, the situation may in fact be more complex:

> Studies have shown that there is very little correspondence between the ability to write a program and the ability to read one. Both need to be taught along with some basic test and debugging strategies. (Winslow, 1996, p. 21)

2.4. Procedural Versus Object-Oriented

A number of recent studies explore issues relating to the object-oriented (OO) programming paradigm (e.g., C++, Java), particularly in contrast to the most common procedural paradigm (e.g., Pascal, C). In general such studies should be seen in the context that there is not likely to be any universally "best" programming notation for comprehension, but that a given notation may assist the comprehension of certain kinds of information by highlighting it in some way in the program code (Gilmore & Green, 1984).

Détienne (1997) reviews claims regarding the "naturalness, ease of use, and power" of the OO approach. Such claims are based on the argument that objects are natural features of problem domains, and are represented as explicit entities in the programming domain, so the mapping between domains is simple and should support and facilitate OO design/programming. The papers reviewed do not support this position.[3] They show that identifying objects is not an easy process, that objects identified in the problem domain are not necessarily useful in the program domain, that the mapping between domains is not straightforward, and that novices need to construct a model of the procedural aspects of a solution in order to properly design objects/classes. While the literature on expert programmers is more supportive of the naturalness and ease of OO design it also shows that expert OO programmers use both OO and procedural views of the programming domain, and switch between them as necessary (Détienne, 1997). Similarly Rist (1995) describes the relationship between plans (a fundamental unit of program design, as discussed above) and objects as "orthogonal".

> Plans and objects are orthogonal, because one plan can use many objects and one object can take part in many plans. (Rist, 1995, pp. 555–556)

Rist (1996) suggests that OO programming is not different, "it is more", because OO design adds the overheads of class structure to a procedural system.

Two recent studies have explored the problems encountered by novices in detail. Wiedenbeck et al. (1999) studied the comprehension of procedural and OO programs in subjects in their second semester of study at university. Subjects were learning either Pascal or C++, and were tested on programs written in the language they were learning (but carefully designed so that

[3]Note that in all studies reviewed by Détienne the novice OO programmers had previous experience in procedural programming, and are therefore not necessarily equivalent to completely novice programmers.

versions in each language were equivalent). For short programs (one class in C++) there was no significant difference in overall comprehension between languages, though the OO subjects were better specifically at understanding the function of the program. Results were completely different when longer programs (multiple classes) were used, with procedural programmers doing better than OO programmers on all measures. The authors conclude that:

> The distributed nature of control flow and function in an OO program may make it more difficult for novices to form a mental representation of the function and control flow of an OO program than of a corresponding procedural program . . . (Wiedenbeck et al., 1999, p. 276)

> We tend to believe that the comprehension difficulties that novices experienced with a longer OO program are attributable partly to a longer learning curve of OO programming and partly to the nature of larger OO programs themselves. (Wiedenbeck et al., 1999, p. 277)

This view does not support the claim that the OO paradigm is a “natural” way of conceptualising and modelling real world situations:

> These results suggest that the OO novices were focusing on program model information, in opposition to claims that he OO paradigm focuses the programmer on the problem domain by modeling it explicitly in the program text. (Wiedenbeck et al., 1999, p. 274)

Similar conclusions are reached by Wiedenbeck and Ramalingam (1999) in a study of C++ students comprehending small programs in C and C++. Once again no difference in overall measures of comprehension were found. Comparing specific measures, however, suggested that subjects tend to develop representations of (small) OO programs that are strong with respect to program function, but weaker with respect to control flow and other program related knowledge. In contrast subjects' representations of procedural programs were stronger in program related knowledge. Results for the better performing half of subjects were then compared to those of the worse performing half. For the better performing group no difference was found. All differences between the OO and procedural conditions were attributable to the worse performing subjects. Burkhardt, Détienne, and Wiedenbeck (1997) proposed a theory of OO program comprehension (including the models constructed by programmers and the effect of expertise on the construction of models) within which many of these factors can be explored.

2.5. Other

A range of other topics have been addressed. Early studies in particular explored particular kinds of language structure or notation (such as the use of GOTOs vs. nested if-then-else structures), various elements of programming practice (such as flow charting and code formatting), and common tasks such as debugging and testing – see, for example, the review in Sheil (1981).

Bishop-Clark (1995) reviews studies of the effects of cognitive style and personality on programming. While no clear trends emerge Bishop-Clark suggests that the common use of a single “unitary” measure of programming success (such as a score or grade) may obscure more subtle effects which could be revealed by studies that relate style and personality to “four stages of computer programming”, namely problem representation, design, coding and debugging.

3. NOVICE PROGRAMMERS

From our perspective as teachers we are most interested in the question of how novices learn to program. This area of interest is set in the general context of cognitive psychology, and topics such as knowledge representation, problem solving, working memory, and so on.

> [Our review] highlights the approaches to understanding human cognition which are of special relevance to programming research. Concepts that recur in many cognitive theories include schemas, production systems, limited resources, automation of skills with practice, working memory, semantic networks and mental models. Most employ propositional representations of one form or another, in which information is represented at a symbolic level. (Ormerod, 1990, p. 77)

Readers unfamiliar with this background can find an introduction in texts such as Anderson (2000).

We now explore topics relating to novice programming in more depth, particularly with respect to program generation. In the context of the literature reviewed above studies of novices and of program generation are in the minority. Even so they form a sizeable body of work, in particular the papers collected in Soloway and Spohrer (1989) *Studying the novice programmer* are a major resource.

3.1. The Task

Learning to program is not easy. In a good overview of what is involved du Boulay (1989) describes five overlapping domains and potential sources of difficulty that must be mastered. These are: (1) general *orientation*, what programs are for and what can be done with them; (2) the *notional machine*, a model of the computer as it relates to executing programs; (3) *notation*, the syntax and semantics of a particular programming language; (4) *structures*, that is, schemas/plans as discussed above; (5) *pragmatics*, that is, the skills of planning, developing, testing, debugging, and so on.

> None of these issues are entirely separable from the others, and much of the 'shock' [. . .] of the first few encounters between the learner and the system are compounded by the student's attempt to deal with all these different kinds of difficulty at once. (du Boulay, 1989, p. 284)

Rogalski and Samurçay summarise the task as follows:

> Acquiring and developing knowledge about programming is a highly complex process. It involves a variety of cognitive activities, and mental representations related to program design, program understanding, modifying, debugging (and documenting). Even at the level of computer literacy, it requires construction of conceptual knowledge, and the structuring of basic operations (such as loops, conditional statements, etc.) into schemas and plans. It requires developing strategies flexible enough to derive benefits from programming aids (programming environment, programming methods). (Rogalski & Samurçay, 1990, p. 170)

Green (1990, p. 117) suggests that programming is best regarded not as "transcription from an internally held representation", or in the context of "the pseudo-psychological theory of 'structured programming' ", but as an exploratory process where programs are created "opportunistically and incrementally". A similar conclusion is reached by Visser (1990) and by Davies:

> . . . emerging models of programming behavior suggest an incremental problem-solving process where strategy is determined by localized problem-solving episodes and frequent problem re-evaluation. (Davies, 1993, p. 265)

An emphasis on opportunistic exploration seems particularly appropriate when considering novice programming.

3.2. Mental Models and Processes

Writing a program involves maintaining many different kinds of “mental model” (see, e.g., Johnson-Laird, 1983), quite apart from a model/knowledge of the programming language itself.

Programs are usually written for a purpose – with respect to some task, problem, or specification. Clearly an understanding/mental model of this problem domain must precede any attempt to write an appropriate program, see, for example, Brooks (1977, 1983), Spohrer, Soloway, and Pope (1989), Davies (1993), Rist (1995). Taking this point to its logical conclusion Deek, Kimmel, and McHugh (1998) describe a first year computer science course based on a problem solving model, where language features are introduced only in the context of the students’ solutions to specific problems.

Other important mental models can be identified. Many studies have noted the central role played by a model of (an abstraction of) the computer, often called a “notional machine” (Cañas, Bajo, & Gonzalvo, 1994; du Boulay, 1989; du Boulay, O’Shea, & Monk, 1989; Hoc & Nguyen-Xuan, 1990; Mayer, 1989; Mendelsohn, Green, & Brna, 1990).

> The notional machine an idealized, conceptual computer whose properties are implied by the constructs in the programming language employed. (du Boulay et al., 1989, p. 431)

That the notional machine is defined with respect to the language is an important point, the notional machine underlying Pascal is very different from the one underlying Prolog.

The purpose of the notional machine is to provide a foundation for understanding the behaviour of running programs.

> [a major issue] is the need to present the beginner with some model or description of the machine she or he is learning to operate via the given programming language. It is then possible to relate some of the troublesome hidden side-effects to events happening in the model, as it is these hidden, and visually unmarked, actions which often cause problems for beginners. However, inventing a consistent story that describes events at the right level of detail is not easy. (du Boulay, 1989, pp. 297–298)

du Boulay et al. (1989) suggest that to be useful the notional machine should be simple, and supported with some kind of concrete tool which allows the model to be observed. In short, a “glass box” instead of a “black box”.

The programmer must also develop a design/model of the program itself and how it will run.

> A running program is a kind of mechanism and it takes quite a long time to learn the relation between a program on the page and the mechanism it describes. (du Boulay, 1989, p. 285)

du Boulay likes building a model of a program based on the program text trying to understand how a car engine works based on a diagram of the engine. The task is much complicated by the many different ways of viewing a program, such as linear order, control flow, data flow, modular structure, or possibly object based structure (see, e.g., Rist, 1995). Corritore and Wiedenbeck (1991) showed that novices (comprehending short Pascal segments) had more difficulty with data flow and function/purpose questions than with control flow, and had least problems with "elementary operations" such as assignment to a variable. Wiedenbeck, Fix, and Scholtz (1993) describe expert mental models of computer programs as founded on the recognition of basic patterns/schemas which are hierarchical and multi-layered, with explicit mappings between layers, well connected internally, and well founded in the program text. Novice representations generally lacked these characteristics, but in some cases were working towards them.

Complicating this picture still further, we suggest, is the distinction between the model of the program as it was intended, and the model of the program as it actually is. Designs can be incorrect, unpredicted interactions can occur, bugs happen. Consequently, programmers are frequently faced with the need to understand a program that is running in an unexpected way. This requires the ability to track or "hand trace" code to build a model of the program an predict its behaviour (which Perkins, Hancock, Hobbs, Martin, and Simmons (1989) call "close tracking" and describe as "taking the computer's point of view"). The process of building such a model (which itself supposes models of both the features of the language and the behaviour of the machine) is a central part of program comprehension, and of the planning, testing and debugging involved in program generation.

Some bugs are minor and can be fixed without change to the program model. In situations where diagnosing a bug exposes a flaw in the underlying model, however, debugging the code may result in major conceptual changes. Pennington and Grabowski (1990) state that diagnosis is the most difficult aspect of debugging, with subsequent corrections being (at least in the case of simple programs where a large re-design is not required) comparatively easier.

Gray and Anderson (1987) call alterations to program code “change episodes”, and suggest that they are rich in information, helping to reveal the programmers models, goals and planning activities.

3.3. Novice Capabilities and Behaviour

Novices lack the specific knowledge and skills of experts, and this perspective pervades much of the literature. Various studies as reviewed by Winslow (1996) concluded that novices are: limited to surface knowledge (and organise knowledge based on superficial similarities); lack detailed mental models; fail to apply relevant knowledge; use general problem solving strategies (rather than problem specific or programming specific strategies); and approach programming “line by line” rather than at the level of meaningful program “chunks” or structures. In contrast to experts, novices spend very little time planning. They also spend little time testing code, and tend to attempt small “local” fixes rather than significantly reformulating programs (Linn & Dalbey, 1989). They are frequently poor at tracing/tracking code (Perkins et al., 1989). Novices can have a poor grasp of the basic sequential nature of program execution: “What sometimes gets forgotten is that each instruction operates in the environment created by the previous instructions” (du Boulay, 1989, p. 294). Their knowledge tends to be context specific rather than general (Kurland, Pea, Clement, & Mawby, 1989). There is no evidence that learning programming fosters an improvement in general problem solving skills, although it may improve (or in turn be improved by prior experience with) very closely related skills such as translating word problems into equations (Mayer, Dyck, & Vilberg, 1989).

Some of this rather alarming list relates to aspects of knowledge, and some to strategies. Perkins and Martin (1986) note that “knowing” is not necessarily clear cut, and novices that appear to be lacking in certain knowledge may in fact have learned the required information (e.g., it can be elicited with hints). They characterise knowledge that a student has but fails to use as “fragile”. Fragile knowledge may take a number of forms: missing (forgotten), inert (learned but not used), or misplaced (learned but used inappropriately). Strategies can also be fragile, with students failing to trace/track code even when aware of the process (see also Davies, 1993; Gilmore, 1990b).

Several studies that focus on novices’ understanding and use of specific kinds of language feature are presented in Soloway and Spohrer (1989). Samurçay (1989) explores the concept of a variable, showing that initialisation is a complex cognitive operation with reading (external input) better understood than assignment (see also du Boulay, 1989). Updating and testing variables seemed to be of roughly equivalent complexity, and were better understood than initialisation. Hoc (1989) showed that certain kinds of abstractions can lead to errors in the use of conditional tests. In a study of bugs in simple Pascal programs (which read some data and perform some processing in the mainline) Spohrer et al. (1989) found that bugs associated with loops and conditionals were much more common that those associated with input, output, initialisation, update, syntax/block structure, and overall planning. Soloway, Bonar, and Ehrlich (1989) studied the use of loops, noting that novices preferred a "read then process" rather than a "process then read" strategy. du Boulay (1989) notes that "for" loops are problematic because novices often fail to understand that "behind the scenes" the loop control variable is being updated. "This is another example of the ubiquitous problem of hidden, internal changes causing problems" (du Boulay, 1989, p. 295). du Boulay also notes problems that can arise with the use of arrays, such as confusing an array subscript with the value stored. Kahney (1989) showed that users have a variety of (mostly incorrect) approximate models of recursion. Similarly, Kessler and Anderson (1989) found that novices were more successful at writing recursive functions after learning about iterative functions, but not vice versa. Issues relating to flow of control were found to be more difficult than other kinds of processing. Many of the points summarised here are also addressed by Rogalski and Samurçay (1990). Détienne (1997) summarises some problems that are specific to OO programmers, including a tendency to think that instance objects are created automatically, and misconceptions about inheritance.

As well as these language feature specific problems there are more general misconceptions. "The notion of the system making sense of the program according to its own very rigid rules is a crucial idea for learner to grasp" (du Boulay, 1989, p. 287). In this respect anthropomorphism ("it was trying to...", "it thought you meant...") can be misleading. Similarly, novices know how they intend a given piece of code to be interpreted, so they tend to assume that the computer will interpret it in the same way (Spohrer & Soloway, 1989). Although prior knowledge is of course an essential starting point, there are times when analogies applied to the new task of programming can also be misleading. Bonar and Soloway (1989) develop this point, exploring the role of existing knowledge (e.g., of step-by-step processes), natural language, and analogies based on these domains as a source of error.

For example, some novices expect, based on a natural language interpretation, that the condition in a "while" loop applies continuously rather than being tested once per iteration.

The underlying cause of the problems faced by novices is their lack of (or fragile) programming specific knowledge and strategies. While the specific problems noted above are significant, some have suggested that this lack manifests itself primarily as problems with basic planning and design. Spohrer and Soloway (1989), for example, collected data in a semester long introductory Pascal programming course (taught at Yale University). Discussing two "common perceptions" of bugs, the authors claim that:

> Our empirical study leads us to argue that (1) yes, a few bug types account for a large percentage of program bugs, and (2) no, misconceptions about language constructs do not seem to be as widespread or as troublesome as is generally believed. Rather, many bugs arise as a result of *plan composition problems* – difficulties in putting the pieces of the program together [. . .] – and not as a result of *construct-based problems*, which are misconceptions about language constructs. (Spohrer & Soloway, 1989, p. 401)

Spohrer and Soloway describe nine kinds of plan composition problem (some of which we have already touched on above):

(a) *Summarisation problem.* Only the primary function of a plan is considered, implications and secondary aspects may be ignored.
(b) *Optimisation problem.* Optimisation may be attempted inappropriately.
(c) *Previous-experience problem.* Prior experience may be applied inappropriately.
(d) *Specialisation problem.* Abstract plans may not be adapted to specific situations.
(e) *Natural-language problem.* Inappropriate analogies may be drawn from natural language.
(f) *Interpretation problem.* "Implicit specifications" can be left out, or "filled in" only when appropriate plans can be easily retrieved.
(g) *Boundary problem.* When adapting a plan to specific situations boundary points may be set inappropriately.
(h) *Unexpected cases problem.* Uncommon, unlikely, and boundary cases may not be considered.
(i) *Cognitive load problem.* Minor but significant parts of plans may be omitted, or plan interactions overlooked.

Spohrer et al. (1989) found a common source of error was “merged plans”, where the same piece of code is intended by the programmer to implement two plans/processes which should have been implemented separately. Often one crucial subplan or step is omitted.

While specific problem taxonomies could be debated (and are likely influenced by language, task, and context) the underlying claim is important – basic program planning rather than specific language features is the main source of difficulty. A similar conclusion is reached by Winslow (1996):

> [An important point] is the large number of studies concluding that novice programmers know the syntax and semantics of individual statements, but they do not know how to combine these features into valid programs. Even when they know how to solve the problems by hand, they have trouble translating the hand solution into an equivalent computer program. (Winslow, 1996, p. 17)

Winslow focuses specifically on the creation of a program rather than the underlying problem solving, noting, for example, that most undergraduates can average a list of numbers, but less than half of them can write a loop to do the same operations. Rist (1995) makes the same point in a different way, summarising the concept of a “focus” (also known as a key or beacon). A focus is the single step (or line) which is the core operation in a plan (or program).

> Focal design [. . .] occurs when a problem is decomposed into the simplest and most basic action and object that defines the focus of the solution, and then the rest of the solution is built around the focus. Essentially, the focus is where you break out of theory into action, out of the abstract into the concrete level of design. (Rist, 1995, p. 537)

To restate the above discussion in these terms, the most basic manifestation of novices’ lack of relevant knowledge and strategies is evident in problems with focal design.

Finally, Rogalski and Samurçay (1990) make an interesting claim (which we have not seen repeated elsewhere).

> Studies in the field and pedagogical work both indicate that the processing dimension involved in programming acquisition is mastered best. The representation dimension related to data structuring and problem modeling is the ‘poor relation’ of programming tasks. (Rogalski & Samurçay, 1990, p. 171)

This would be an interesting topic to pursue further. It may not be the case that the "processing dimension" is any easier to master, but rather that problem modelling and representation are logically prior, so that novices who are experiencing problems manifest them at that early stage, while those who are working successfully progress through both representation and processing tasks.

3.4. Kinds of Novice

While much attention has been paid to the study of novices versus experts, it is clear that it is also useful to explore the topic of novices versus novices. A group of novices learning to program will typically contain a huge range of backgrounds, abilities, and levels of motivation, and also typically result in a huge range of unsuccessful to successful outcomes. As we might expect, measures of general intelligence are related to success at learning to program (Mayer et al., 1989). As noted above (Section 2.5), however, Bishop-Clark (1995) found no clear trends emerging from a review of studies of the effects of cognitive style and personality on programming. Rountree, Rountree, and Robins (2002) found that from a survey (covering factors such as background, intended major, expected workload and so on) of students in an introductory programming paper, the most reliable predictor of success was the grade that the student expected to achieve. This and other results showed that students in general have a reasonably accurate sense of how they are likely to do within the first 2 weeks of the course.

Despite the fact that it is apparently not measured by or significant in the cognitive style and personality tests used so far, different kinds of characteristic behaviour are certainly evident when observing novices in the process of writing programs. Perkins et al. (1989) distinguish between two main kinds, "stoppers" and "movers". When confronted with a problem or a lack of a clear direction to proceed, stoppers (as the name implies) simply stop. "They appear to abandon all hope of solving the problem on their own" (Perkins et al., 1989, p. 265). Student's attitudes to mistakes/errors are important. Those who are frustrated by or have a negative emotional reaction to errors are likely to become stoppers. Movers are students who keep trying, experimenting, modifying their code. Movers can use feedback about errors effectively, and have the potential to solve the current problem and progress. However, extreme movers, "tinkerers", who are not able to trace/track their program, can be making changes more or less at random, and like stoppers have little effective chance of progressing.

4. NOVICE LEARNING AND TEACHING IN CS1

4.1. Goals and Progress

Most novices learn to program via formal instruction such as a computer science introductory course ("CS1"). This sets the topic of novice learning and teaching in the context of an extensive educational literature. Current theory suggests a focus not on the instructor teaching, but on the student learning, and effective communication between teacher and student. The goal is to foster "deep" learning of principles and skills, and to create independent, reflective, life-long learners. The methods involve clearly stated course goals and objectives, stimulating the students' interest and involvement with the course, actively engaging students with the course material, and appropriate assessment and feedback. For a good introduction see, for example, Ramsden (1992).

Teaching standards clearly influence the outcomes of courses that teach programming (Linn & Dalbey, 1989). Linn and Dalbey propose a "chain of cognitive accomplishments" that should arise from ideal computer programming instruction. This chain starts with the *features of the language* being taught. The second link is *design skills*, including templates (schemas/plans), and the procedural skills of planning, testing and reformulating code. The third link is *problem-solving skills*, knowledge and strategies (including the use of the procedural skills) abstracted from the specific language taught that can be applied to new languages and situations. This chain of accomplishments forms a good summary of what could be meant by deep learning in introductory programming.

Given the goals of deep learning an observation that recurs with depressing regularity, both anecdotally and in the literature, is that the average student does not make much progress in an introductory programming course. Exploring roughly semester long courses in middle schools, Linn and Dalbey note that few students get beyond the language features link of the chain, and conclude that "the majority of students made very limited progress in programming" (Linn & Dalbey, 1989, p. 74). A study of students with 2 years of programming instruction (Kurland et al., 1989) concludes on a similar note, that "many students had only a rudimentary understanding of programming". Winslow observes that "One wonders [...] about teaching sophisticated material to CS1 students when study after study has shown that they do not understand basic loops..." (Winslow, 1996, p. 21). Soloway, Ehrlich, Bonar, and Greenspan (1983), for example, studied students who had completed a single semester programming course. When asked to write a loop which calculated an average (excluding a sentinel value signalling the end of input) only 38% were able to complete the task correctly (even when syntax errors were ignored).

4.2. Course Design and Teaching Methods

For the moment we will assume a conventionally structured course based on lectures and practical laboratory work, and a conventional curriculum focused largely on knowledge – particularly relating to the features of the language being taught and how to use them. Why is it that most introductory programming courses and textbooks adopt this approach? Obvious reasons include the important role of such knowledge in programming and the sheer volume and detail of language related features that can be covered. More subtly, as Brooks (1990) points out, while the use of strategies strongly impacts on the final program that is produced, the strategies themselves cannot (in most cases) be deduced from the final form of the program. Finished example programs are rich sources of information about the language which can be presented, analysed and discussed. The strategies that created those programs, however, are much harder to make explicit.

Ideally course design and teaching would take place in the context of familiarity with the key issues that have been identified in the literature. The most basic factor, especially given the observations regarding the limited progress made by novices in introductory courses, is that a CS1 course should be realistic in its expectations and systematic in its development: "Good pedagogy requires the instructor to keep initial facts, models and rules simple, and only expand and refine them as the student gains experience" (Winslow, 1996, p. 21). du Boulay et al. (1989) make a case for the use of simple, specially designed teaching languages. In many cases the role of the course in the broader teaching curriculum may rule this out as an option, and complex "real" languages are typically used.

A major recommendation to emerge from the literature is that instruction should focus not only on the learning of new language features, but also on the combination and use of those features, especially the underlying issue of basic program design.

> From our experience [. . .] we conclude that students are not given sufficient instruction in how to "put the pieces together." Focusing explicitly on specific strategies for carrying out the coordination and integration of the goals and plans that underlie program code may help to reverse this trend. (Spohrer & Soloway, 1989, pp. 412–413)

A further important suggestion is to address the kinds of mental models which underlie programming:

> Models are crucial to building understanding. Models of control, data structures and data representation, program design and problem domain are all important. If the instructor omits them, the students will make up their own models of dubious quality. (Winslow, 1996, p. 21)

Two specific points have been tested by Mayer (1989). Mayer showed that students supplied with a notional machine model (which Mayer called a "concrete model") were better at solving some kinds of problem than students without the model. Mayer also showed, as we would predict from the general educational literature, that students who are encouraged to actively engage and explore programming related information (by paraphrasing/restating it in their own words) performed better at problem solving and creative transfer (see also Hoc & Nguyen-Xuan, 1990).

With particular reference to OO programming Wiedenbeck and Ramalingam (1999, p. 84) summarise the pedagogical implications of their study. The authors suggest that the OO style aids the understanding of program function for small programs, but that – especially as programs grow in size – particular attention should be paid to control flow and data flow in teaching, and the use of aids to comprehension.

The laboratory based programming tasks that are part of a typical CS1 course have some pedagogically useful features. Each one can form a "case based" problem solving session. The feedback supplied by compilers and other tools is immediate, consistent, and (ideally) detailed and informative. The reinforcement and encouragement derived from creating a working program can be very powerful. In this context students can work and learn on their own and at their own pace, and "programming can be a rich source of problem-solving experience" (Linn & Dalbey, 1989, p. 78). Working on easily accessible tasks, especially programs with graphical and animated output, can be stimulating and motivating for students. However such tasks should still be based on and emphasise the programming principles that underlie the effects (Kurland et al., 1989). Especially in the context of practical tasks, paired or collaborative work and "peer learning" has also been shown to be beneficial (Van Gorp & Grissom, 2001; Williams, Wiebe, Yang, Ferzli, & Miller, 2002; Wills, Deremer, McCauley, & Null, 1999).

Soloway and Spohrer (1989, p. 417) summarise several suggestions relating to the design of development environments/programming tools that support novices. These include: the use of "graphical languages" to make control flow explicit; a simple underlying machine model; short, simple and consistent naming conventions; graphical animation of program states (with no "hidden" actions or states); design principles based on spatial metaphors; and the gradual withdrawal of initial supports and restrictions. Anderson and colleagues (Anderson, Boyle, Corbett, & Lewis, 1990; Anderson, Boyle, Farrell, & Reiser, 1987; Anderson, Conrad, & Corbett, 1989) have developed an extensive and effective intelligent tutoring system for LISP within the ACT* model of learning and cognition (Anderson, 1983, 1990).

Finally for a broad perspective, offered in respect to teaching Java but which could equally apply to any kind of educational situation, Burton suggests that teachers keep in mind the distinctions between "what actually gets taught; what we think is getting taught; what we feel we'd like to teach; what would actually make a difference" (Burton, 1998, p. 54).

4.3. Alternative Methods and Curricula

Some recommendations regarding the teaching of programming suggest a fundamental change in the focus of CS1 teaching, to the extent that if fully implemented they would represent alternative kinds of curricula.

An important recommendation noted above is that instruction should address the underlying issue of basic program design, in particular the use of the schemas/plans which are the central feature of programming knowledge representation. Such an emphasis could be accommodated within a conventional curriculum, or could form the basis of an alternative approach.

> Explicit naming and teaching of basic schemata [...] may become part of computer programming curricula. (Mayer, 1989, p. 156)

> ...students should be made aware of such concepts as goals and plans, and such composition statements as abutment and merging [...]. We are suggesting that students be given a whole new vocabulary for learning how to construct programs. (Spohrer & Soloway, 1989, p. 413)

Soloway and Ehrlich (1984) explored this approach as a basis for teaching Pascal. Similar ideas regarding the identification and teaching of solutions to particular classes of programming problems can be found in the OO "patterns" literature, see, for example, Gamma, Helm, Johnson, and Vlissides (1994). For an analysis and overview of the use of pattern languages for teaching see Fincher (1999a), and for two recent descriptions of courses based on patterns see Reed (1998) and Proulx (2000).

Is it effective to teach schemas directly to novices, rather than expect them to emerge from examples and experience? Some general support is provided from a review of mechanisms of skill transfer (see, e.g. Robins, 1996), but transfer and analogical mechanisms are complex. Deep, structural similarities are often not identified and exploited. While supporting the idea of teaching schemas Perkins et al. (1989) also suggest that alternative methods may be more generally effective:

> Instruction designed to foster bootstrap learning but not providing an explicit schematic repertoire might produce competent and flexible programmers, and might yield the broad cognitive ripple effects some advocates of programming instruction have hoped for. (Perkins et al., 1989, p. 277)

From a theoretical perspective, in some accounts of learning and knowledge consolidation such as Anderson's influential ACT family of models (Anderson, 1976, 1983, 1993), abstract representations of knowledge cannot be learned directly. They can be only learned "by doing", that is, by practicing the operations on which they are based.

Problem solving has also been identified as a possible foundation for teaching programming. Fincher (1999b) argues in favour of problem solving based teaching, and categorises and briefly reviews the related "syntax free", "literacy" and "computation-as-interaction" approaches. Deek et al. (1998) describe a first year computer science course based on a problem solving model, where language features are introduced only in the context of the students' solutions to specific problems. In this environment students in the problem solving stream generally rated their own abilities and confidence slightly more highly than did students in the control stream (receiving traditional instruction). Students in the problem solving stream also achieved a significantly better grade for the course (with e.g. an increase from 5% to over 25% of the students attaining "A" grades). An extensive discussion of the practical issues involved in problem based learning, a description of problem based learning courses, and a 3-year longitudinal follow-up of students is described in Kay et al. (2000).

Like schema/pattern based methods the problem solving based approaches clearly have promise. However as noted (Section 3.3) by for example Winslow (1996) and Rist (1995), problem solving is necessary, but not sufficient, for programming. The main difficulty faced by novices is expressing problem solutions as programs. Thus the coverage of language features and how to use and combine them must remain an important focus.

For an influential and completely different perspective on the art of teaching programming Dijkstra (1989), in the evocatively titled "On the cruelty of really teaching computer science", argues that anthropomorphic metaphors, graphical programming environments and the like are misleading and represent an unacceptable "dumbing down" of the process. Dijkstra proposes a very different kind of curriculum based on mathematical foundations such as predicate calculus and Boolean algebra, and establishing formal proofs of program correctness. (A lively debate ensues in the subsequent peer commentary.)

While it is clear that alternatives to conventional curricula show promise, it is also the case that none of them has come to dominate the theory or practice of programming pedagogy. Most textbooks, for example, are still based on a conventional curriculum model. In future work we intend to review and assess the literature on these alternative methods and their effectiveness.

5. DISCUSSION

5.1. Summary and Implications

In this section we briefly summarise the material reviewed above, and highlight some of the most important points and practical implications for teachers. The summary follows the structure of the paper (hence e.g. "2.3:" refers to Section 2.3) and the implications noted here can be directly supported by the literature reviewed in the relevant section.

The psychological/educational literature relating to programming is large and complex. A number of trends can be identified. 2.1: The first is a distinction between novices and experts, with an emphasis on the many deficits of novices. 2.2: The second trend is the distinction between knowledge and strategies. An important though ill-defined concept is the schema/plan as the most important building block of programming knowledge. An important but open question is why and how different strategies emerge, and how these are related to underlying knowledge. 2.3: The third trend is the distinction between program comprehension and generation, with models of the former being particularly numerous. When generating programs novices must create their program plans (experts can often retrieve them), hence explicit attention (in course design and teaching) to planning and problem solving may be beneficial. Clearly these aspects of programming are related, with comprehension playing an important role in supporting generation, but there is some suggestion that individuals' abilities with respect to these tasks may not be well correlated. This has implications for course design and assessment – comprehension based assessment tasks may not be a good measure of the ability to write programs. 2.4: The fourth, recent trend, is a comparison of OO and procedural programming styles. There is little support for the claim that the OO approach allows for significantly easier modelling of problem domains, with both OO design and traditional procedural factors identified as significant. Hence even in OO based courses it may be necessary, particularly for weaker students, to devote particular attention to procedural concepts, flow of control, flow of data, and design (see also 4.2).

In this literature the majority of studies focus on program comprehension, often in experts, and typically based on experimental studies. Our focus on this paper has been on novices, particularly novice program generation, and in the process by which this is taught and learned. 3.1: It is clear that novice programmers face a very difficult task. Learning to program involves acquiring complex new knowledge and related strategies and practical skills. Hence initial course material should be simple, and this should be expanded on systematically as the students gain experience (see also 4.2). 3.2: Novice programmers must learn to develop models of the problem domain, the notional machine, and the desired program, and also develop tracking and debugging skills so as to model and correct their programs. Explicitly identifying and addressing each of these topics may be beneficial. In practical/ laboratory based work it may be useful for instructors to particularly attend to change episodes (where students alter their code), as these may be rich in information about the students' models, plans and goals. 3.3: Novices typically have many deficits in both knowledge and strategies. Familiarity with the specific issues identified in the literature may aid course design. Loops, conditionals, arrays and recursion have all been identified as language features that are especially problematic, and could benefit from particular attention. Several authors have suggested, however, that the most important deficits relate to the underlying issues of problem solving, design, and expressing a solution/design as an actual program. The frequent practical programming exercises that are a feature of most programming courses are almost certainly central in addressing this issue, and it may also help to encourage in practical work (e.g., in the design of laboratory workbooks and the like) the use of an explicit software development method to give some structure to the process.[4] Novice's problems exacerbated by the fact that where knowledge and strategies are learned, they are often fragile (not applied, or misapplied). Further research may be useful here to ascertain whether this is a deficit in accessing learned material, recognising the situations in which it is appropriate, or having the confidence to use it/ experiment. 3.4: Different kinds of characteristic novice behaviour can be identified, including movers, stoppers, and tinkerers. Where such distinctions are thought to apply to a given student, this may help to effectively target the assistance given.

With respect to teaching novices in CS1 type courses the goal is to foster deep learning in students. 4.1: Many students make very little progress in a first programming course. 4.2: Various suggestions regarding course design and delivery have been made in the literature. Most of these, such as paying particular attention to issues of basic design, have already arisen/been noted in the summary above. Other suggestions relate to the design of development environments/programming tools for teaching novices. It may be helpful to make aspects of control flow and data flow explicit, and avoid "hidden" actions or states. 4.3: Finally, course designs based on explicitly teaching schemas, problem solving, and mathematical foundations have been proposed.

5.2. A Programming Framework

One summary of topics relating to novice programming makes explicit the implied relationships between many of the issues. This structural summary is outlined in the "programming framework" shown in Figure 1.

The framework highlights as one dimension the individual attributes of the programmer, namely their knowledge, strategies, and mental models. The fundamental role of knowledge and strategies was discussed in Section 2.2. The importance of various mental models (e.g., of the notional machine, and of the program) was noted in Section 3.2. The second dimension of the framework separates the phases of designing, generating and evaluating a program. Issues relating to design/planning were reviewed in Section 3.3. The process of program generation is of course central, as described in Sections 2.3 and 3.3. The important role of evaluation (comprehension, tracking, debugging) was noted in Sections 2.3 and 3.2. Combining the dimensions, the

[4]See Koffman and Wolz (2002) for a good example of a textbook which consistently uses such a method.

	Knowledge	**Strategies**	**Models**
Design	of planning methods, algorithm design, formal methods	for planning, problem solving, designing algorithms	of problem domain, notional machine
Generation	of language, libraries, environment / tools	for implementing algorithms, coding, accessing knowledge	of desired program
Evaluation	of debugging tools and methods	for testing, debugging, tracking / tracing, repair	of actual program

Fig. 1. A programming framework. This framework summarises the relationships between a number of issues relating to programming. It should be read mainly by columns, that is, *knowledge* of planning methods (required to *design* a program), *knowledge* of a language (required to *generate* a program), *knowledge* of debugging tools (required to *evaluate* a program), and so on. In many cases we would interpret the "cells" of the framework as "fuzzy", rather than making very sharp distinctions.

overall framework emphasises the fact that the different kinds of individual attributes are not single "undifferentiated" constructs, but are brought to bear (perhaps with varying efficacy) at different stages of the programming process.

Ideally, teaching and learning would take place in the context of familiarity with the main issues that have been identified in the literature. We suggest that any compact summary, such as for example the framework proposed above, could have a variety of uses. For example, it may be helpful during the process of course design to highlight factors which might be incorporated into the course content, delivery and assessment. It may be useful to make it available to students, both as an aid to identifying the learning objectives of the course (which has many advantages, see, e.g., Ramsden, 1992) and as an aid to "metalearning" or "learning about learning" (see, e.g., Vilalta & Drissi, 2002). A summary may also be useful to teachers/teaching assistants involved in practical/laboratory work, as an aid to diagnosing the problems of and assisting individual students. For example, it may be helpful to identify the fact that a given student has a good knowledge of design principles but poor strategies for applying them, or perhaps good strategies for design but poor strategies for debugging/testing. For this latter purpose in particular, any diagnostic tool to be used in an actual laboratory situation will need to be rich enough to be useful, but simple enough to be manageable.

5.3. Comments and Future Work

In this final section we make some more speculative observations, and note possible topics for future work. These comments are based on our experience of the review presented above, and of our own teaching and recent study of the programming course that we teach (Rountree et al., 2002).

From our point of view as teachers there is a distinction which is much more important than the one between novices and experts which has received so much attention in the literature. This is the distinction between *effective* and *ineffective* novices. Effective novices are those that learn, without excessive effort or assistance, to program. Ineffective novices are those that do not learn, or do so only after inordinate effort and personal attention. It may be productive, in an introductory programming course, to explicitly focus on trying to create and foster effective novices. In other words, rather than focusing exclusively on the difficult end product of programming knowledge, it may be useful to focus at least in part on the enabling step of functioning as an effective novice.

What underlying properties make a novice effective? How can we best turn ineffective novices into effective ones? A deeper understanding of both kinds of novices is required. The range of potentially relevant factors includes motivation, confidence or emotional responses, and aspects of general or specific knowledge, strategies, or mental models.

As a first step towards addressing these questions, we further suggest that the most significant differences between effective and ineffective novices relate to strategies rather than knowledge. Language related knowledge is available from many sources, and courses and textbooks are typically designed to introduce this knowledge in a structured way. The strategies for accessing this knowledge and applying it to program comprehension and generation, however, are crucial to the learning outcome, but typically receive much less attention. What are the strategies employed by effective novices, how do they relate to their knowledge and their relevant mental models, and can these strategies be taught to ineffective novices?

Others have also suggested that strategies are central. Perkins et al. note that “certain broad attitudes and conducts” characterise unsuccessful novices:

> . . . behaviors such as stopping, neglect of close tracking, casual tinkering, and neglect of or systematic errors in breaking problems down. (Perkins et al., 1989, p. 277)

These are all deficits in strategy. Davies states that:

> Even in the case of novice programmers we have seen that the strategic elements of programming skill may, in some cases, be of greater significance than knowledge-based components. (Davies, 1993, p. 265)

We would go so far as to say *especially* in the case of novice programmers, and in *most* rather than some cases. Given that knowledge is (assumed to be) uniformly low, it is their preexisting strategies that initially distinguish effective and ineffective novices.

> As novices do not have the specialised knowledge and skills of the expert, one might expect their performance to be largely function of how well they can bring their skills from other areas to bear. (Sheil, 1981, p. 119)

> . . . youngsters vary widely in their progress, succeeding only to the extent that they happen to bring with them the characteristics that make them good bootstrap learners in the programming context. (Perkins et al., 1989, pp. 277–278)

Differences in initial strategies will interact with other factors, such as motivation and the capacity to acquire language related knowledge, to rapidly separate novices along the effective-ineffective continuum.

What are the implications of this view? We suggest that programming strategies should receive more and more explicit attention in introductory programming courses. One way to address this would be to introduce many examples of programs as they are being developed (perhaps "live" in lectures), discussing the strategies used as part of this process.[5] As well as needing to know more about effective an ineffective novice, we need to know how to foster effective strategies in all novices through course design and delivery. We need to motivate students, engage them in the process, and make them want to learn to be effective programmers.

In future work we intend to further explore these issues and to focus on the topic of novice strategies. Why do many novices, even when aware of the techniques and encouraged to use them, fail to plan their programs? What are the main reasons why many students become so consistently stuck, and can these be diagnosed and addressed? What is the relationship between the

[5]As noted above (Section 4.2), example programs are sources of programming knowledge, but the strategies that go into creating a program are not usually visible in the final product.

ability to generate and the ability to comprehend a program? Are strategy deficits generic or related to an inability to construct or maintain a mental model of the program? What kind of support will best address the needs of each kind of novice? How can we present language related knowledge so as to best develop and foster appropriate strategies and models? Perhaps one of the most important aspects to be explored is why relevant knowledge and strategies are often known but not applied. Finally, of course, the underlying issue is how best to use the answers to such questions to better teach and foster the learning of novice programmers.

ACKNOWLEDGMENTS

This work has been supported by internal University of Otago Research into Teaching grants. We are also grateful for the support and suggestions of our many colleagues. Special thanks to Richard O'Keefe, and to other members of the COMP103 teaching and support team including Natalie Adams, Michael Atkinson, Tracey Cuthbertson, Sandy Garner, Parviz Najafi and many other committed and enthusiastic lab demonstrators.

REFERENCES

Anderson, J.R. (1976). *Language, memory and thought.* Hillsdale, NJ: Erlbaum Associates.

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

Anderson, J.R. (1993). *Rules of the mind.* Hillsdale, NJ: Erlbaum.

Anderson, J.R. (2000). *Cognitive psychology and its implications* (5th ed.). New York: Worth Publishing.

Anderson, J.R., Boyle, C., Corbett, A., & Lewis, M.W. (1990). Cognitive modeling and intelligent tutoring. *Artificial Intelligence*, *42*, 7–49.

Anderson, J.R., Boyle, C., Farrell, R., & Reiser, B. (1987). Cognitive principles in the design of computer tutors. In P. Morris (Ed.), *Modeling cognition* (pp. 93–134). NY: Wiley.

Anderson, J.R., Conrad, F.G., & Corbett, A.T. (1989). Skill acquisition and the LISP tutor. *Cognitive Science*, *13*, 467–505.

Bishop-Clark, C. (1995). Cognitive style, personality, and computer programming. *Computers in Human Behavior*, *11*, 241–260.

Boehm, B.W. (1981). *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall.

Bonar, J., & Soloway, E. (1989). Preprogramming knowledge: A major source of misconceptions in novice programmers. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 324–353). Hillsdale, NJ: Lawrence Erlbaum.

Brooks, F.P., Jr. (1995). *The mythical man-month: Essays on software engineering anniversary edition.* Reading, MA: Addison-Wesley.

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

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

Brooks, R.E. (1990). Categories of programming knowledge and their application. *International Journal of Man-Machine Studies*, *33*, 241–246.

Burkhardt, J., Détienne, F., & Wiedenbeck, S. (1997). Mental representations constructed by experts and novices in object-oriented program comprehension. In S. Howard, J. Hammond, & G. Lindgaard (Eds.), *Human-computer interaction: INTERACT'97* (pp. 339–346). London: Chapman & Hall.

Burton, P. (1998). Kinds of language, kinds of learning. *ACM SIGPLAN Notices*, *33*, 53–61.

Cañas, J.J., Bajo, T., & Gonzalvo, P. (1994). Mental models and computer programming. *International Journal of Human-Computer Studies*, *40*, 795–811.

Corritore, C.L., & Wiedenbeck, S. (1991). What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, *3*, 199–222.

Davies, S.P. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies*, *39*, 237–267.

Deek, F.P., Kimmel, H., & McHugh, J.A. (1998). Pedagogical changes in the delivery of the first-course in computer science: Problem solving, then programming. *Journal of Engineering Education*, *87*, 313–320.

Détienne, F. (1990). Expert programming knowledge: A schema based approach. In J.M. Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 205–222). London: Academic Press.

Dijkstra, E.W. (1989). On the cruelty of really teaching computer science. *Communications of the ACM*, *32*, 1398–1404.

Dreyfus, H., & Dreyfus, S. (1986). *Mind over machine: The power of human intuition and expertise in the era of the computer.* New York: Free Press.

du Boulay, B. (1989). Some difficulties of learning to program. In E. Soloway & J.C. Spohrer (Eds.), (pp. 283–299). Hillsdale, NJ: Lawrence Erlbaum.

du Boulay, B., O'Shea, T., & Monk, J. (1989). The black box inside the glass box: presenting computing concepts to novices. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 431–446). Hillsdale, NJ: Lawrence Erlbaum.

Fincher, S. (1999a). Analysis of design: An exploration of patterns and pattern languages for pedagogy. *Journal of Computers in Mathematics and Science Teaching: Special Issue CS-ED Research*, *18*, 331–348.

Fincher, S. (1999b). What are we doing when we teach programming? *29th ASEE/IEEE Frontiers in Education Conference*, 12a4-1–12a4-5.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software.* Reading, MA: Addison-Wesley.

Gilmore, D.J. (1990a). Methodological issues in the study of programming. In J.M. Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 83–98). London: Academic Press.

Gilmore, D.J. (1990b). Expert programming knowledge: A strategic approach. In J.M. Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 223–234). London: Academic Press.

Gilmore, D.J., & Green, T.R.G. (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, *21*, 31–48.

Gray, W.D., & Anderson, J.R. (1987). Change-episodes in coding: When and how do programmers change their code? In G.M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers: Second Workshop* (pp. 185–197). Norwood, NJ: Ablex.

Green, T.R.G. (1990). Programming languages as information structures. In J.M. Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 117–137). London: Academic Press.

Green, T.R.G., Bellamy, R.K.E., & Parker, J.M. (1987). Parsing and ginsarp: A model of device use. In H.J. Bullinger & B. Shackel (Eds.), *Proceedings INTERACT'87.* Amsterdam: Elsevier/North-Holland.

Guindon, R. (1990). Knowledge exploited by experts during software systems design. *International Journal of Man-Machine Studies*, *33*, 182–279.

Hoc, J.M. (1989). Do we really have conditional statements in our brains? In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 179–190). Hillsdale, NJ: Lawrence Erlbaum.

Hoc, J.M., Green, T.R.G., Samurçay, R., & Gillmore, D.J. (Eds.). (1990). *Psychology of programming.* London: Academic Press.

Hoc, J.M., & Nguyen-Xuan, A. (1990). Language semantics, mental models and analogy. In J.M. Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 139–156). London: Academic Press.

Humphrey, W.S. (1999). *Introduction to the team software process.* Reading, MA: Addison-Wesley/Longman.

Johnson-Laird, P.N. (1983). *Mental models.* Cambridge: Cambridge University Press.

Kahney, H. (1989). What do novice programmers know about recursion? In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 209–228). Hillsdale, NJ: Lawrence Erlbaum.

Kay, J., Barg, M., Fekete, A., Greening, T., Hollands, O., Kingston, J., & Crawford, K. (2000). Problem-based learning for foundation computer science courses. *Computer Science Education*, *10*, 109–128.

Kessler, C.M., & Anderson, J.R. (1989). Learning flow of control: Recursive and iterative procedures. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 229–260). Hillsdale, NJ: Lawrence Erlbaum.

Koffman, E., & Wolz, U. (2002). *Problem solving with Java* (2nd ed.). Reading, MA: Addison-Wesley.

Kurland, D.M., Pea, R.D., Clement, C., & Mawby, R. (1989). A study of the development of programming ability and thinking skills in high school students. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 83–112). Hillsdale, NJ: Lawrence Erlbaum.

Letovsky, S. (1986). Cognitive processes in program comprehension. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers, First Workshop* (pp. 58–79). Norwood, NJ: Ablex.

Linn, M.C., & Dalbey, J. (1989). Cognitive consequences of programming instruction. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 57–81). Hillsdale, NJ: Lawrence Erlbaum.

Mayer, R.E. (1989). The psychology of how novices learn computer programming. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 129–159). Hillsdale, NJ: Lawrence Erlbaum.

Mayer, R.E., Dyck, J.L., & Vilberg, W. (1989). Learning to program and learning to think: what's the connection? In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 113–124). Hillsdale, NJ: Lawrence Erlbaum.

Mendelsohn, P., Green, T.R.G., & Brna, P. (1990). Programming languages in education: The search for an easy start. In J.M. Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 175–199). London: Academic Press.

Mills, H.D. (1993). Zero defect software: Cleanroom engineering. *Advances in Computers*, *36*, 1–41.

Ormerod, T. (1990). Human cognition and programming. In J.M. Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 63–82). London: Academic Press.

Pennington, N. (1987a). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, *19*, 295–341.

Pennington, N. (1987b). Comprehension strategies in programming. In G.M. Olson, S. Sheppard, & E. Soloway (Eds.), *Empirical studies of programmers: Second Workshop* (pp. 100–112). Norwood, NJ: Ablex.

Pennington, N., & Grabowski, B. (1990). The tasks of programming. In J.M. Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 45–62). London: Academic Press.

Perkins, D.N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1989). Conditions of learning in novice programmers. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 261–279). Hillsdale, NJ: Lawrence Erlbaum.

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

Perlis, A., Sayward, F., & Shaw, M. (1981). *Software metrics: An analysis and evaluation.* Cambridge, MA: MIT Press.

Proulx, V.K. (2000). Programming patterns and design patterns in the introductory computer science course. *Proceedings of the thirty-first SIGCSE technical symposium on computer science education* (pp. 80–84). New York: ACM Press.

Ramsden, P. (1992). *Learning to teach in higher education.* London: Routledge.

Reed, D. (1998). Incorporating problem-solving patterns in CS1. *SIGCSE Bulletin*, *30*, 6–9.

Rist, R.S. (1986a). Plans in programming: Definition, demonstration and development. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers, First Workshop.* Norwood, NJ: Ablex.

Rist, R.S. (1986b). Focus and learning in program design. *Proceedings of the 8th Annual Conference of the Cognitive Science Society* (pp. 371–380). Hillsdale, NJ: Lawrence Erlbaum.

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

Rist, R.S. (1990). Variability in program design: The interaction of process with knowledge. *International Journal of Man-Machine Studies*, *33*, 305–322.

Rist, R.S. (1995). Program structure and design. *Cognitive Science*, *19*, 507–562.

Rist, R.S. (1996). Teaching Eiffel as a first language. *Journal of Object-Oriented Programming*, *9*, 30–41.

Robins, A. (1996). Transfer in cognition. *Connection Science*, *8*, 185–203.

Rogalski, J., & Samurçay, R. (1990). Acquisition of programming knowledge and skills. In J.M. Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 157–174). London: Academic Press.

Rountree, N., Rountree, J., & Robins, A. (2002). Identifying the danger zones: Predictors of success and failure in a CS1 course. *Inroads (the SIGCSE Bulletin)*, *34*, 121–124.

Sackman, H. (1970). *Man-computer problem solving.* Princeton, NJ: Auerbach.

Samurçay, R. (1989). The concept of variable in programming: Its meaning and use in problem solving by novice programmers. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 161–178). Hillsdale, NJ: Lawrence Erlbaum.

Sheil, B.A. (1981). The psychological study of programming. *Computing Surveys*, *13*, 101–120.

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

Soloway, E., Adelson, B., & Ehrlich, K. (1988). Knowledge and processes in the comprehension of computer programs. In M. Chi, R. Glaser, & M. Farr (Eds.), *The nature of expertise* (pp. 129–152). Hillsdale, NJ: Lawrence Erlbaum.

Soloway, E., Bonar, J., & Ehrlich, K. (1989). Cognitive strategies and looping constructs. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 191–207). Hillsdale, NJ: Lawrence Erlbaum.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *SE-10*, 595–609.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1983). What do novices know about programming? In B. Shneiderman & A. Badre (Eds.), *Directions in human-computer interactions* (pp. 27–54). Norwood, NJ: Ablex.

Soloway, E., & Spohrer, J.C. (Eds.). (1989). *Studying the novice programmer.* Hillsdale, NJ: Lawrence Erlbaum.

Spohrer, J.C., & Soloway, E. (1989). Novice mistakes: Are the folk wisdoms correct? In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 401–416). Hillsdale, NJ: Lawrence Erlbaum.

Spohrer, J.C., Soloway, E., & Pope, E. (1989). A goal/plan analysis of buggy Pascal programs. In E. Soloway & J.C. Spohrer (Eds.), *Studying the novice programmer* (pp. 355–399). Hillsdale, NJ: Lawrence Erlbaum.

van Dijk, T.A., & Kintsch, W. (1983). *Strategies of discourse comprehension.* New York: Academic Press.

Van Gorp, M.J., & Grissom, S. (2001). An empirical evaluation of using constructive classroom activities to teach introductory programming. *Computer Science Education*, *11*, 247–260.

Vilalta, R., & Drissi, Y. (2002). A perspective view and survey of meta-learning. *Artificial Intelligence Review*, *18*, 77–95.

Visser, W. (1990). More or less following a plan during design: Opportunistic deviations in specification. *International Journal of Man-Machine Studies*, *33*, 247–278.

Visser, W., & Hoc, J.M. (1990). Expert software design strategies. In J.M. Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (Eds.), *Psychology of programming* (pp. 235–250). London: Academic Press.

von Mayrhauser, A., & Vans, A.M. (1994). *Program understanding – A survey* (Tech. Rep. CS-94-120). Department of Computer Science, Colorado State University.

Weinberg, G.M. (1971). *The psychology of computer programming.* New York: Van Nostrand Reinhold.

Widowski, D., & Eyferth, K. (1986). Comprehending and recalling computer programs of different structural and semantic complexity by experts and novices. In H.P. Willumeit (Ed.), *Human decision making and manual control* (pp. S.267–275). Amsterdam: North-Holland Elsevier.

Wiedenbeck, S., Fix, V., & Scholtz, J. (1993). Characteristics of the mental representations of novice and expert programmers: An empirical study. *International Journal of Man-Machine Studies*, *25*, 697–709.

Wiedenbeck, S., & Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies*, *51*, 71–87.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C.L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, *11*, 255–282.

Williams, L., Wiebe, E., Yang, K., Ferzli, M., & Miller, C. (2002). In support of pair programming in the introductory computer science course. *Computer Science Education*, *12*, 197–212.

Wills, C.E., Deremer, D., McCauley, R.A., & Null, L. (1999). Studying the use of peer learning in the introductory computer science curriculum. *Computer Science Education*, *9*, 71–88.

Winslow, L.E. (1996). Programming pedagogy – A psychological overview. *SIGCSE Bulletin*, *28*, 17–22.