In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of 

The University of British Columbia
Vancouver, Canada

Date 

92-06-27

DE-6 (2/88)
ABSTRACT

Research into the teaching of computer programming seldom examines the profound and abundant knowledge of classroom practitioners. Further, little research has been carried out which compares the learning and teaching of procedural programming to the learning and teaching of other programming paradigms.

In this study, teachers of post-secondary computer programming were interviewed in order to gain an understanding of the ways in which their knowledge about programming and their knowledge about teaching programming are related and in order to get a sense of the conceptual frameworks within which their knowledge is formed.

The results of the study indicate that knowledge about programming and knowledge about teaching programming are strongly related. The respondents' teaching and programming experience, the social contexts within which they operate, their desire for control, their programming paradigm experience and the mental models they employ when programming shape the ways in which they program and the ways in which they teach.

All of the respondents feel that languages and paradigms affect the way programmers think when programming. As well, those with broad experience feel that because there are large differences between paradigms, shifting from one paradigm to another is very difficult. Thus they feel that the choice of students' first programming language is critical.

There appears to be a strong relationship between respondents' paradigm experience and the mental models they use. Those respondents with experience in multiple
paradigms tend to use more abstract models than those who have been exposed primarily to the procedural paradigm. These more experienced respondents feel that they are able to choose languages that match the way they think.

The results of the study imply that investigations of programming education research and practice can benefit from an understanding of researchers' and practitioners' frameworks. In particular, it appears that the choice of programming paradigm strongly influences research studies and classroom teaching.
CONTENTS

LIST OF TABLES viii
LIST OF FIGURES ix
ACKNOWLEDGEMENT x

CHAPTER ONE: INTRODUCTION 1
  Purpose of the Study 2
  Need for the Study 4
  Generalizability 4
  Research Questions 5
  Terminology and Style 6

CHAPTER TWO: REVIEW OF THE LITERATURE 7
  Computer Science Education 7
    A Focus on Programming 7
      What is Programming? 8
        A Classical View 8
        An Adaptive View 8
        A Mathematical View 9
      Summary 10
    What do Programmers Do? 10
  Why Is Programming Taught? 13
    Professional Training 13
    Computer Literacy 13
    Cognitive Skills Development 14
    Summary 15
  Current Issues 15
    Predicting Success 15
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Respondents' Paradigms</td>
<td>39</td>
</tr>
<tr>
<td>2</td>
<td>Interview Protocol</td>
<td>59</td>
</tr>
<tr>
<td>3</td>
<td>Respondents' Life Experience</td>
<td>148</td>
</tr>
<tr>
<td>4</td>
<td>The Relationship of Mental Model to Respondents' Views</td>
<td>164</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Programming Paradigms</td>
<td>29</td>
</tr>
<tr>
<td>2</td>
<td>Programming Paradigms</td>
<td>37</td>
</tr>
<tr>
<td>3</td>
<td>Lorne’s Conceptual Model</td>
<td>85</td>
</tr>
<tr>
<td>4</td>
<td>Breadth of Respondents’ Paradigm Experience</td>
<td>156</td>
</tr>
<tr>
<td>5</td>
<td>Respondents’ Mental Models</td>
<td>161</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENT

The process of creating this document has re-enforced the feeling that my ideas are shaped and clarified through interaction with others. Many thanks to those who shared in the process: to the respondents for sharing their worlds; to the members of my committee, Dr. Marvin Westrom, Dr. Gaalen Erickson and Dr. Hugh Dempster, for their advise and encouragement; to my fellow grad students for their opinions and companionship -- in particular to Tony for exemplifying dedication to a goal and to Ken for reminding me that the side paths are just as important; to my colleagues, whose anecdotes sparked the idea for this research; to my family for their patience and support; and to Bruce, for listening and questioning and pushing me through.
CHAPTER ONE

INTRODUCTION

Research into factors affecting the acquisition of programming skills is predominantly quantitative in nature (Ellis, 1989). Moreover, a broadening of the scope of the research to an examination of literature about learning outcomes, teaching strategies and choice of programming language reveals the same quantitative tone throughout the field (Sharma, 1987; Van Merrienboer and Krammer, 1987; Johanson, 1988).

The statistical tenor of the literature represents the intrinsically quantifiable nature of computer programming. The current study might have taken the same perspective had not the quantitative nature of the research contrasted so strongly with the qualitative nature of the educational research being carried out in other content areas such as science and mathematics (Yackel, Cobb, Wood, Wheatley and Merkel, 1990; Nussbaum and Novick, 1982; Posner, Strike, Hewson and Gertzog, 1982; Driver, 1989; Jacob, 1988); with the view that “truth in the fields of human affairs is better approximated by statements that are rich with the sense of human encounter ...” (Stake, 1978, p. 6).

Teachers of programming, in numerous papers, express opinions about predicting success, cognitive outcomes, appropriate teaching strategies and best choice of programming language (e.g., Bork, 1971; Hancock, 1988; Luker, 1989; Shneiderman, 1985; Johanson, 1988). Their opinions, based on “perceptions and understanding” (Stake, 1978, p. 6), represent profound and abundant insights into the teaching of programming that is not being captured by the plethora of quantitative analysis.

What Teachers Know about Programming
Purpose of the Study

The purpose of this study is to extend the field of computer programming education research by adopting a form of "educational connoisseurship" (Eisner, 1985, p. 216); by taking "the opportunity to attend to happenings of educational life in a focused, sensitive, and conscious way" (p. 221). In particular, the study examines computer programming education from a constructivist perspective.

To the constructivist, knowledge is a human construction which enables one to cope with the world (von Glasersfeld, 1989; Driver, 1989). Moreover, the human construction does not correspond with an objective truth. Knowledge is nothing more or less than a subjective fit with experience. Public knowledge is a social construction which represents a consensus rather than a recognition of existing facts. Thus, computer scientists, teachers, students and researchers all have personal interpretations of the socially constructed concept of programming. The constructivist postulates the existence of conceptual frameworks which allow people to deal with events. Driver and Erickson (1983) define a conceptual framework as "the mental organisation imposed by an individual on sensory inputs as indicated by regularities in an individual's responses to particular problem settings" (p. 39). "Individual's actions", they maintain, "are influenced by their conceptual structures" (p. 39). From a constructivist perspective, then, the existing quantitative research into computer programming education is missing an important facet: an understanding of social and personal interpretations of programming, and a sense of the conceptual frameworks which influence teachers' pedagogical practices.

In this study, in the style of studies of teachers' views in areas such as curriculum planning (Elbaz, 1983), curriculum change (Cardwell, 1988) and occupational patterns (Lortie, 1975), I examine the views of programming teachers. In order to
recast the issues extant in the literature in an interpretive framework, I attempt to make explicit teachers' constructed knowledge about programming and teaching programming and examine the factors which may have affected that construction. The constructivist would agree with Bogdan and Taylor that "to understand the workings of a group ... we must also understand how they ... define their world ... " (in Spector, 1984, p. 460). In this study, in order to gain an understanding of how programming teachers define their world, I interviewed, in depth, six programming teachers.

The notion that it is necessary to get a sense of teachers' definitions of the world points to another salient aspect of the existing computer programming education literature. As the field of computer science matures, distinct programming language paradigms are emerging (Appleby, 1991). Computer science educators are just beginning to address these distinctions and to come to grips with a professional paradigm shift away from the conventional procedural style (Carey and Shepherd, 1988; Luker, 1989; Skublics and White, 1991; Wells and Kurtz, 1989).

Because programming languages "[influence] our thinking habits" (Dijkstra, in Rich, 1984, p. 7), and "[serve] as a framework within which we organize our ideas about processes" (Abelson and Sussman, 1985, p. 4), the constructivist perspective would suggest a close look at the role of programming language paradigms in shaping teachers' interpretations of programming and teaching programming.

In this study, therefore, respondents who have experience in using and teaching various programming styles were deliberately selected. The study focuses on the post-secondary training of computer professionals in an attempt to capture the views of teachers who are most likely to be cognizant of paradigmatic differences.
Similarities and differences between respondents are analysed with particular attention to the paradigms with which they are familiar.

Need for the Study

The results of the existing quantitative research in the area of computer programming education are controversial and contradictory (Sharma, 1987; Finlay, 1988; Cafolla, 1988; Oman, 1986; Sorge and Wark, 1984; Johanson, 1988; Lemos, 1975; Palumbo, 1990; Ellis, 1989). Studies of factors affecting the acquisition of programming skills yield contradictory results (Finlay, 1988; Oman, 1986; Ellis, 1989). The contention that programming may be successfully used to teach problem solving has found little supporting evidence (Johanson, 1988; Palumbo, 1990). Controversy surrounds the discussion of best first programming language (Johanson, 1988; Luker, 1989). Several authors have offered explanations for the uncertain results, citing reasons such as poor research methods and varying instructional styles (e.g. Johanson, 1988; Salomon and Perkins, 1987; Palumbo, 1990), but none have attempted to capture the underlying interpretations of programming implicit in the literature.

In order to understand the emerging role of programming paradigms, it is necessary to examine implicit interpretations of programming. Researchers must be aware of the underlying interpretations held by their subjects and by themselves. They must make explicit the programming paradigms being used. An exploration of the knowledge constructed by practitioners is a starting point for investigation into these interpretations. An understanding of teachers' knowledge and the factors which affect the construction of that knowledge will allow us to take a more systematic approach to research in computer science education; to make better sense
of the discussions of cognitive outcomes, prediction of success, choice of teaching strategy and choice of programming language.

Generalizability

As Erickson (1986) explains, the concept of generalizability takes on a new meaning for the qualitative researcher. "The search is not for abstract universals arrived at by statistical generalization from a sample population, but for concrete universals, arrived at by studying a specific case in great detail and then comparing it with other cases studied in equally great detail" (p. 130, emphasis in original). This study generalizes in the sense that it is "epistemologically in harmony with the reader's experience and thus to that person a natural basis for generalization" (Stake, 1980, p. 64). Furthermore, as Eisner (1985) states, "[what] we can productively ask of a set of ideas is not whether it is really true but whether it is useful, ... , whether it enables one to perceive the phenomenon in more complex and subtle ways ..." (p. 241, emphasis in original).

Research Questions

This study sets out to examine the articulated perceptions of post-secondary computer programming teachers in order to address the following questions:

How do teachers organize their knowledge about programming?

How do teachers organize their knowledge about teaching programming?

How do teachers' constructions about programming integrate with their constructions about teaching programming?
Terminology and Style

The field of computer programming makes use of terms that may be unfamiliar to the reader. Technical terms have been defined in a glossary in Appendix A.

To differentiate them from textual quotations, transcript excerpts have been presented in italics without quotation marks.
CHAPTER TWO

REVIEW OF THE LITERATURE

Computer Science Education

Computer science education is a very young discipline. The first high-level languages were defined in the 1950s and computing education has become prevalent only in the last twenty years (MacGregor, 1988; Sleeman, Putnam, Baxter and Kuspa, 1986; McIntyre, 1991). Despite the relative immaturity of the field, the computer as an object of and an aid to learning has been the subject of considerable research and debate.

This section examines current thinking about the learning of programming: why it is learned; how it is learned; and who is most likely to succeed at learning it.

A Focus on Programming

Programming is a major component of post secondary computer science curricula (Ralston and Shaw, 1980; Koffman, Miller and Wardle, 1984; Denning et al., 1989). It is introduced early in the curriculum (usually in the first course (Hirshfield, 1985)) and Wells and Kurtz (1989) go so far as to say that “most people equate computer science with programming” (p. 246). Since programming appears to be central to the discipline, much discussion has focused on the acquisition of programming skills.

Before examining various aspects of the programming discussion, it is necessary to investigate the frameworks within which the discussion is occurring. What is meant by the term programming, what is it that programmers do, and why do we include programming in our curricula?
What is Programming?

Webster’s Third New International Dictionary defines a computer program as “a sequence of coded instructions for a digital computer”. A programmer is one who works out the sequence of instructions. Programming, then, is the act of writing the instructions. Despite the simplicity of the definition, interpretations of the term are many.

A Classical View.

The classical view of programming is articulated in Appleby’s (1991) recent computer science textbook. The practice of programming, she says, “consists of program design, coding, documentation, and debugging” (p. 3). This definition has been instilled in the minds of computer science educators by way of the ACM recommended curricula (Denning et al., 1989).

An Adaptive View.

The classical view contrasts strongly with that of Weinberg (1971) who, in examining the psychology of computer programming, finds that “programming ... is a single word that encompasses an infinitude of activities” (p. 121). He claims that the classical definition, even when extended to include problem definition and analysis “distorts the truth” (Weinberg, p. 133) in three ways. “First of all, the sequence is not so fixed .... Secondly, not all steps need be present .... Thirdly, it need not be a sequence at all ...” (Weinberg, p. 133, emphasis in original). The classical decomposition, he claims, is too refined. “[The] divisions lack sharp boundaries, or perhaps have no boundaries at all” (Weinberg, p. 133). Sheil (1981) also takes issue with the prevailing definition of programming. In a review of the psychological research on programming, he claims that existing research takes “a shallow view of
the nature of programming skill" (p. 101). He suggests that programming consists of a diverse collection of complex learned skills supported by an elaborate knowledge base.

Weinberg's and Sheil's early rather holistic views of programming are echoed by Karp (1989), who speaks of "the inevitable intertwining of software development with specification" (p. 1411). He quotes a colleague, Richard Fateman, who says that "programming ... is more often a process of discovery and adaptation, as various relevant components of a problem are examined in turn, and a goal, initially specified incompletely, is reached" (p. 1411).

Soloway (1988), retaining the adaptive flavour of Fateman's definition, goes further by redefining programming. He says,

> The term programming is neither accurate nor illuminating: "programming" comes from "program," which is the final product of the synthesis process. Programming does not convey the type of knowledge and skills that need to go into the process of developing that product, that artifact (p. 127).

By programming, he means "the learning and use of synthesis skills" (p. 122).

A Mathematical View.

Dijkstra's view differs from both the classical and the adaptive views. In 1989 he "challenged some of the basic assumptions on which [computer science] curricula are based" (Denning, 1989, p. 1397) in a talk entitled "On the Cruelty of Really Teaching Computer Science". To Dijkstra, "the programmer's ... main task is to give a formal proof that the program he proposes meets the equally formal functional specification" (Dijkstra, 1989, p. 1404). His interpretation seems to be that programming is mainly a formalized task akin to that of the mathematician or logician. Several of the esteemed respondents to Dijkstra's recommendations agree,
in principle, with his point of view (Denning, 1989). For example, David Parnas (1985), who resigned from the Strategic Defense Initiative Organization because of his lack of trust in the ability of programmers to provide for integrity in such a system, shares Dijkstra's opinions about verifiability. Conventional programming methods, he claims, "[lead] to confusion and [result] in systems that nobody can understand completely" (p. 1330).

Summary.

Computer science educators ascribe differing meanings to the term 'programming'. The classical view is widely held, although seldom explicitly stated. The adaptive view, although delineated by various researchers, does not seem to have made a powerful impact on the way in which programming is taught (Palumbo, 1990). Neither does Dijkstra's mathematical view appear to have strongly affected computer science education (Oliver and Malone, 1991; McIntyre, 1991).

What do Programmers Do?

Although the three perspectives articulated above differ substantially, there appears to be agreement that programming involves a process: a sequence of well-defined steps, an adaptive learning process or a formalized methodology. What does that process entail? Various studies have attempted to determine what programmers are doing when they program.

Research indicates that novices' views of programming differ greatly from experts' views (Kurland, Pea, Clement and Mawby, 1989; Mayer, 1989b; McKeithen, Reitman, Rueter and Hirtle, 1981). Novices tend to hold a surface view of programming, relying on format and syntactic properties of code, whereas experts rely on a more functional organization based on underlying structures of problems and solution
strategies. In programming studies analogous to the classic study of expert and
novice chess players (Chase and Simon, 1973), experienced programmers could
recall more lines of code from a working program than could novices. However
experts and novices performed equally when the program was made up of random
statements (Mayer, 1989b; McKeithen et al., 1981). Other research shows that there
do exist in novices categories of alternative frameworks or "underlying problems"
(Spohrer and Soloway, 1989, p. 411) that are independent of programming language
(Joni and Soloway, 1986; Pea, 1986; Soloway, Ehrlich, Bonar and Greenspan, 1982;
Sleeman et al., 1986; Lee and Lehrer, 1988; Scherz, Goldberg and Fund, 1990; Putnam,
Sleeman, Baxter and Kuspa, 1989). Some argue, then, that expert programmers have
well-developed frameworks not shared by novices which, for them, provide
working interpretations of computer programs.

What is it that expert programmers are doing that differentiates them from novices?
Schwartz's (1975) list of recommendations concerning what programmers should
know may shed light on this question. Firstly, he says, programmers should
understand algorithms. That is, they should recognize and understand "important"
(p. 21) algorithms, by which he means well-known ones. Programmers should also
understand that some processes cannot be performed by a program and that some
can be performed only very inefficiently. Secondly, programmers should
understand "semantic frameworks, which allow individual algorithms to be
organized into large program structures" (p. 21). Thirdly, Schwartz feels that
programmers should have "a conscious view of the programming process," and
should "understand the way in which programs, in their earliest origins, coalesce
out of less organized intellectual structures" (p. 21). Schwartz's recommendations
allude to deep underlying structures which shape the way in which expert
programmers work. Expert programmers are operating at a level much deeper than
that at which novices are working. In order to inform the discussion of expert/novice differences, Palumbo (1990) and Shneiderman and Mayer (1979) differentiate between syntactic and semantic knowledge. Palumbo states that experts have "two major attributes: a solid command of the factual, declarative, or linguistic base necessary to solve the problems presented; and a collection of effective strategies for employing that base when solving related problems" (p. 70).

Pennington (1985), delves more deeply into expert programmers' frameworks by examining professional programmers' mental representations of programs. She differentiates between procedural abstraction and functional abstraction. A programmer using procedural abstraction will segment a program into groupings which reflect the control structures of the programming language. The program is segmented based on clues in the language syntax. The programmer using functional abstraction, on the other hand, sees patterns in the code which represent stereotypical functional units. The segmentation reflects the dataflow structure of the program, rather than the control flow. That is, functional abstraction depends on what is happening in the program, rather than on how it is happening. The programmer is able to segment the program based on partial pattern matches to known functional units. Pennington found that most programmers used procedural abstraction. She also found a difference between COBOL and Fortran programmers, leading her to wonder if the programming language may affect the difficulty of extracting information from the program text. She does not consider that perhaps COBOL and Fortran programmers conceptualize programming in different ways.
Thus research shows that expert programmers make use of a broad base of syntactic knowledge and an organized framework which allows them to use that knowledge base to solve problems.

**Why Is Programming Taught?**

There are three primary reasons for which programming is included in school curricula: the training of programming professionals, computer literacy and the development of thinking skills (Salomon and Perkins, 1987; Becker, 1986; Schultz and Hart, 1983).

**Professional Training.**

Programming is taught, in an industrial setting and at the post-secondary level, for the purpose of training professional programmers, scientists and engineers -- those who would program computers to perform specific job-related tasks. Despite the fact that a decreasing demand for professional programmers is predicted with the introduction of each new high-level tool, from the creation of the first high-level languages in the 1950s to the recent widespread acceptance of Computer Assisted Software Engineering (CASE) tools, industry demand for entry-level programmers continues to grow (March, 1989; Rifkin, 1989). Thus professional training continues to be a major goal of programming instruction, particularly at the post-secondary level.

**Computer Literacy.**

With the increased acceptance and availability of computers in the early 1980s, computer literacy for all students became the goal of many curricula. The prevailing view was that in order to be computer literate, one needed to be able to program (Kay, 1989). This view of computer literacy led to the inclusion of a large
component of programming in introductory computing courses taught to non-computer science majors at secondary and post-secondary levels. Although "the programming outlook might be outdated" (Kay, p. 37) and despite the sense that "the programming approach has been superseded" (Kay, p. 37) by computer applications, many schools continue to teach programming in literacy courses (Computer Studies 11 Curriculum Guide and Resource Book 1984; Becker, 1986).

Cognitive Skills Development.

A more widespread justification for the teaching of programming to all students is "its possible impact on generalizable cognitive skills" (Salomon & Perkins, 1987, p. 149). As Linn (1985) points out, "the avowed purpose of most programming courses is to teach problem solving or reasoning as well as to teach programming" (p. 14). Papert (1987), an early proponent of teaching programming, says that programming experiences can provide the learner with new ways of thinking. For him, the computer's "deepest role is cultural rather than instrumental. What is important ... is not that the child drew a circle, ... but that this way of working ... provides new ways to think about circles, and through them, new ways to think about mathematics more generally" (p. 29). Feurzeig (1988), in a discussion of the teaching of mathematics in the context of programming, states that "programming provides highly motivated models for fostering the development of the principal heuristic and metacognitive skills" (pp. 102-103). Despite these widespread claims, Liao and Bright (1991), in a meta-analysis of existing research, found only slightly positive results in terms of the effect of learning programming on students' cognitive skills. Educational researchers, rather than questioning the premise that teaching programming can potentially affect thinking skills, tend to express the opinion that results have not been forthcoming because of the way in which programming has
been taught in the past or because of problems with the way in which the research has been conducted (Johanson, 1988).

Summary.

Programming continues to be included in curricula for three primary purposes, then: the training of programming professionals, the education of computer literate users and the development of students' thinking skills. The purpose for which programming is taught, as well as the interpretation of what the programming process involves, will fundamentally affect the nature of programming education. Nonetheless, most authors writing about teaching programming do not explicitly describe what they mean by programming nor why they think it ought to be learned.

Current Issues

Several bodies of research have taken shape within the computer science education literature. Extensive research has been done in the area of predicting success in computer science and, particularly, programming courses. As well, teaching approach, mode of learning, use of models and choice of programming language have been studied.

Predicting Success

Much research has been carried out in the area of programming skill prediction. Ellis (1989) lists 52 studies which were performed at secondary and post-secondary levels in the past twenty years. Predictor variables typically examined may be grouped into four types: prior achievement, aptitude measures, cognitive and psychological measures, and environmental variables.
Prior Achievement.

Grade Point Average (GPA) and mathematics achievement are the most common predictor variables studied (Sall, 1989). Typically predictor studies examine a large number of variables (Greer, 1986; Sorge and Wark, 1984; Finlay, 1988). An exemplar of these studies is that of Petersen and Howe (1979), who examine various environmental variables as well as high school achievement, personality types and general aptitude. They found GPA to be a good predictor of success in introductory computer science as did several others (Ellis, 1989). Prior mathematics achievement, although a reasonably good predictor, does not appear to merit the importance it seems to have acquired as a prerequisite for computer science students (Finlay, 1988; Schulz, 1984).

Aptitude Measures.

A study carried out by Morecroft and Ameen (1987) is representative of the kind of research being done in the area of aptitude (e.g., Guinan and Stephens, 1988; Bitter and Lu, 1988; Oman, 1986). Grades of students majoring in post-secondary level Information Systems were related to Scholastic Aptitude Test (SAT) scores. In addition, students were separated into two groups: those who completed their course of studies and those who did not. Although several studies have shown that SAT scores are a good predictor of programming success (Oman, 1986; Sorge and Wark, 1984; Leeper and Silver, 1982), Morecroft and Ameen find that there is little difference in scores between those who complete their major and those who do not. They conclude, in fact, that the best predictors for success in an Information Systems course of studies are students’ grades in their introductory courses.
Cognitive and Psychological Measures.

A review by Sharma (1987) summarizes investigations into relationships between cognitive style and psychological type variables and performance in computer science courses. Typical cognitive studies examine measures such as analytic-heuristic styles (e.g., Cheney, 1980), analytic and deductive reasoning (e.g., Bitter and Lu, 1988) and field dependence (e.g., Stevens, 1983). Psychological variables studied include Myers-Briggs personality types (e.g., Corman, 1986; Mawhinney and Saraswat, 1991), learning styles (e.g., Corman, 1986) and motivation (e.g., Watson and Behnke, 1991). Although Sharma (1987) concludes that “it is reasonable to deduce that learners' cognitive styles and psychological characteristics should be considered in the planning of programs and strategies” (p. 396), he points out that “there seems to be a general feeling of uncertainty and indecisiveness as to who can benefit from computer-oriented tasks” (p. 397). That is, although there are some promising results arising from the cognitive and psychological studies, there appears to be confusion about the implications and applications of the specific results.

Environmental Variables.

Typical of the environmental variables examined are those investigated by Finlay (1988). She correlates university students' educational backgrounds (i.e. university majors), previous level of mathematical training, previous computer experience, attitude, age and gender to their results in a postgraduate course in Information Processing. Most studies which examine environmental variables correlate them to results in a single course or to students' averages in all coursework (e.g., Greer, 1986; Sorge and Wark, 1984; Oman, 1986). In contrast, Finlay studies the relationship of the variables to each of the courses in the students' program and compares the
results. Interestingly, she finds that factors which predict success in certain programming courses do not predict success in others (p. 40). This result highlights the inherent difficulty in evaluating research without explicit knowledge of the researchers' and educators' tacit understanding of what programming entails, and without explicit information about the sense of programming being understood by the students, that is, its meaning and purpose.

**Summary.**

Ellis (1989) states that the studies he reviewed generally indicate that a student with a strong mathematics and science background, a high grade point average and a high programming aptitude score is likely to succeed in an introductory computer science course (pp. 25-26). However, there is much controversy about which variables most accurately predict programming success (Sharma, 1987; Finlay, 1988; Cafolla, 1988; Oman, 1986; Sorge and Wark, 1984). It appears that much of the confusion and lack of generalizability stems from the fact that researchers typically predict success in programming in general based on results achieved in one specific situation. There exist varied interpretations of the meaning, process and purpose of programming.

**Approaches**

Van Merrienboer and Krammer (1987), in a discussion of instructional design principles for introductory programming courses, outline four categories of skills typically addressed in introductory programming courses: use of the programming environment; programming language syntax; language semantics and analysis; and program generation. They find, however, as do Linn (1985), McIntyre (1991) and Johanson (1988), that “introductory courses often emphasize learning the features [of a language] rather than how to use them to solve problems” (Linn, 1985, p. 15). Throughout the 1970s, programming courses were courses in language syntax. In
the past decade, however, the emphasis in programming courses has been on problem solving rather than on syntax (Van Merrienboer and Krammer, 1987). The difficulty with this approach is that students tend to spend more time coding and debugging than they do planning, designing and testing their algorithms (Van Merrienboer and Krammer, 1987).

The introduction of the concepts of structured programming in the mid-1970s greatly influenced the way in which programming is taught (McIntyre, 1991; Ghezzi and Jazayeri, 1987). The structured approach is so well entrenched that Van Merrienboer and Krammer (1987) “presuppose the use of a structured programming language” (p. 255) in the instructional strategies that they evaluate. They identify three widespread strategies for the teaching of computer science: the expert approach, the spiral approach and the reading approach.

The Expert Approach.

The expert approach is characterized by the immediate introduction to students of non-trivial problems. Students are expected to solve problems in a structured manner. That is, in the design of algorithms and program code, they work within the constraints of a structured language and a set of “discourse rules” (Joni and Soloway, 1986, p. 99) that an expert programmer would use. Typical of the expert approach is the adoption of the ‘stepwise refinement’ model which stresses the top-down design of problem solutions. Students are presented with explicit design diagrams and worked-out examples. Thus students concentrate on planning and high-level semantic content rather than on programming language syntax and basic computer usage. The expert approach, computer science’s counterpart of the scientific method, is widely used and has been the subject of much research and

The Spiral Approach.

The spiral approach, as defined by Shneiderman (1977),

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).

Van Merrienboer and Krammer (1987) define syntactic knowledge as “unorganized knowledge of low level details” (p. 256). Semantic knowledge is “hierarchically organized knowledge” of concepts (p. 256). Using the spiral approach, students are presented with simple ideas in an incremental hierarchical manner in order to foster the development of more complex forms of knowledge. Program generation is stressed. The first problems presented to the students are somewhat trivial, emphasizing syntactic and low-level semantic knowledge. As the students gain competence, more complex specifications are introduced and the emphasis shifts to the design process. The spiral approach does not appear to have been widely adopted in practice (Shneiderman, 1977; Shneiderman and Mayer, 1979; Marchionini, 1985).
The Reading Approach.

The reading or 'whole language' approach "emphasizes the reading, modification and amplification of non-trivial, well-designed and working programs" (Van Merrienboer and Krammer, 1987, p. 257). Students at first observe and evaluate the execution of working programs. They are then introduced to well-structured program code, which they read and trace. The next phase of the reading approach involves the modification of existing programs and practice with simple design and coding. In the final phase, students write their own programs. Hence the students' tasks become more complex as the course progresses while the complexity of the programs presented remains constant. The reading approach has been supported by Bork (1971), Postma and Laing (1981) and Dalbey, Tournaire and Linn (1985).

Summary.

Few reports have been found which compare the success of various approaches to teaching programming. Results of the studies that do exist are conflicting (Lemos, 1975; Postma and Laing, 1981; Dalbey et al., 1985; Van Merrienboer and Krammer, 1987; MacGregor, 1988). Studies of educational approaches normally do not examine implicit views of programming.

Mode of Learning

Clarke (1989), in his discussion of instructional approaches, outlines three modes of learning commonly used in the programming classroom: group interaction, discovery and lecture.
Group Interaction.

As Weinberg (1971) notes, "Programmers do not ordinarily work in isolation. Although an individual programmer may find himself assigned the task of writing a program, even then he has other programmers to whom he may turn for help -- and who, at the same time, may be turning to him" (p. 45). He differentiates between the programming group and the programming team. The latter is a formal group established to work together on a single program. The former is a collection of programmers grouped together informally by virtue of the fact that they work in proximity to one another -- at a school, a computing centre or in a computer department. Interaction between programmers in both circles is important, he finds, in promoting egoless programming, support and creativity.

MacGregor (1988), Lemos (1978) and Webb, Ender and Lewis (1986), among others, studied the effects of group work on programming students' success. All found "positive relationships between group interaction and programming achievement" (MacGregor, 1988, p. 156). In particular, they found that discussion with others allows students to explore their conceptions about the action and outcomes of computer programs. Activities such as hand-tracing (tracing a program on paper), and walkthroughs (verbally tracing program behaviour in a group environment) are used by professional programmers and have been found to be applicable to the classroom environment.

Discovery Learning.

Several educators support the concept of discovery learning. Bork (1971) explains that using this approach, "the student sees no didactic material, encounters no grammatical information, hears no lectures about the language; but he does learn to use it simply by interacting with the language compiler itself" (p. 5). Bork (1971),
Dalbey and Linn (1985) and Shneiderman (1985) all contend that discovery-style learning will enhance students’ success. However, as Dalbey et al. (1985) point out, use of the discovery method alone may preclude instruction in planning skills felt to be essential for advanced programming tasks.

Lecture Mode.

The use of lectures is a common approach to teaching programming. As Clarke (1989) explains, with this approach “all instruction takes place in the lecture room. At the completion of the lecture, the instructor departs and the students move to a laboratory to put the theory into practice” (p. 14). Dalbey and Linn (1985), in a review of various teaching methods, find that this approach is popular often because of practical constraints in the learning environment. Comparisons with other learning techniques show that students prefer situations that promote active student involvement and in which they have more control over their learning (Clarke and Donn, 1990). In a comparison of lecture-style learning to lab-style learning, Clarke and Donn (1990) conclude that a combination of styles may be most appropriate, with syntactical rules being best taught using a lecture method (p. 21).

Summary.

Thus three modes of learning are primarily used when teaching programming: Lecture mode, hands-on learning and group interaction.

Salomon and Perkins (1987) believe that differences in programming instruction are one of the possible reasons some research has yielded significant results in the relationship between programming language instruction and problem-solving skills, whereas other research has not (Palumbo, 1990, p. 78).
Instructional techniques, grounded as they are in understandings of what it means to program, must be explicitly outlined so that valid comparisons of various outcomes can be made.

**Models**

Models and analogies can be helpful in leading students to adopt new conceptual frameworks (Brown and Clement, 1987). It may appear self-evident that the models and analogies used must be plausible to the student, but there are many examples in computer programming of analogies that are useful to experts yet meaningless to students (Lee and Lehrer, 1988; Kahney, 1989; Putnam, et al., 1989; Du Boulay, 1989). A common difficulty with the use of analogy is that experts, because of their conceptual frameworks, see the similarities between the things being compared, whereas students, with different conceptual frameworks, may focus on aspects that are either irrelevant or dissimilar. Thus students who are told that a variable is like a box may assume that a variable may contain more than one value at a time like a box (Putnam et al., 1989). As well, students’ own models and analogies, such as the anthropomorphic model described by Shneiderman (1988), may not be viable. It is therefore important that students become overtly aware of aspects of models and analogies that fit programming processes and those that do not.

Research into the learning of computer programming with and without models (Mayer, 1975; Hancock, 1988; Mayer, 1989a; Bracey, 1989) supports the view that a concrete model enhances students’ understanding of the workings of computer programs. Interestingly, there is some indication that models help lower ability students more than higher ability students and that models "may actually interfere with high ability learners who already have a rich set of more sophisticated knowledge" (Mayer, 1975, p. 733).
Choice of Language

The choice of appropriate programming language has been the subject of a great deal of debate. Perhaps this issue is so contentious because, as Abelson and Sussman (1985) state, "[the programming] language ... serves as a framework within which we organize our ideas about processes" (p. 4). The discussion of programming language parallels the discussion of natural language: "It is widely believed that our thinking is restricted by the expressive power of the language in which we communicate our thoughts. ... Programmers in the process of developing software are similarly constrained" (Sebesta, 1989, p. 2). There are those computer scientists, such as Connors, who claim that a "properly trained programmer thinks primarily in terms of programming, only secondarily in terms of a particular language" (in Rich, 1984, p. 7). On the other hand, there are those who, like Dijkstra, believe that "a programming language ... [exerts] an influence on our thinking habits" (in Rich, 1984, p. 7).

As Sammet (1991) points out,

an exact count of all the high-level languages ever used is impossible even for those used in the United States .... However, since it is possible to name over 1000 that have been used at one time or another since 1953, it is not unreasonable to assume that there are several hundred outside the United States ... (p. 48).

Sammet asks why so many languages have been created and conjectures that there are two main reasons for their proliferation: functional needs and personal needs. She lists several general and specialized application areas which require specific functions. As well, she contends that implementation requirements and size requirements have contributed to the large number of languages in existence. However, she feels that personal needs have been even more influential in the development of languages and cites five of them:
1. personal taste regarding style and syntax,

2. personal opinion regarding language design issues,

3. the fun of language design,

4. the availability of research grants and contracts for the development of new languages, and

5. the ease with which one can publish papers about new language design.

Just as new languages are developed to fulfil personal and functional needs, languages for classroom use are often chosen for similarly practical reasons. Woodhouse (1983) outlines five criteria for the selection of language in programming courses:

1. the availability of a system which can use the language,

2. teacher knowledge,

3. ease of teaching and learning,

4. the functionality of the language, and

5. whether the language is structured.


Not all recommendations about language are based on practicalities, however. Various opinions about the proper language for introductory programming instruction have been expressed, with the most vociferous debate surrounding the
question of whether to teach BASIC or Pascal (Ellis, 1989). Authors advising the use of Smalltalk (Skublics and White, 1991; Borne and Girardot, 1990), PROLOG (Johanson, 1988) and LISP (Grogono, 1989) have recently joined the fray. Unfortunately, little empirical research has been done, leading one to the conclusion that what Sammet (1991) would call personal needs are being used to shape opinion.

It appears that the lack of agreement about which programming language is 'best' for industry and educational use stems from the many differing functional and personal needs perceived by computer scientists and educators. Again the purpose for which the language will be used and the underlying personal understandings of programming will shape the decision about language.

**The Paradigm Shift**

"Never mind the language, what about the paradigm?" asks Luker (1989, p. 252). There are increasing indications that a paradigm shift is occurring in the computer industry and in computer education (Johanson, 1988; Grogono, 1989; Abelson and Sussman, 1985; Skublics and White, 1991; DiNitto, 1988). In the introduction to her text, *Programming languages: Paradigm and practice*, Appleby (1991) outlines Wegner's notion of programming language paradigms. His ideas are an extension of the concept of scientific paradigm introduced by Kuhn (1962). Wegner feels that as computer science matures, distinct language paradigms are emerging. Furthermore, the breakdown of older paradigms can be seen in the current feeling that the computer industry is experiencing a software crisis: software costs are overinflated and the integrity of major software projects is highly questionable (Luker, 1989; Parnas, 1985; Dijkstra, 1989).
Current researchers generally agree upon the following classifications of language (Appleby, 1991; Petre and Winder, 1990; Sebesta, 1989; Wells and Kurtz, 1989):

**Imperative languages:**

Imperative languages imply an underlying machine whose state is manipulated explicitly by program instructions. This type of language is characterized by sequential declaration and assignment instructions, which manipulate the contents of computer memory. The imperative language class may be split into two sub-classes:

A. **Procedural or Block structured languages:** Procedural languages allow for modularization of code in the form of procedures or functions. An example is Pascal.

B. **Object-oriented languages:** Object-oriented languages support interacting objects. All data are objects, all objects are treated uniformly, and all processing is done by passing messages among objects. An example is Smalltalk.

**Declarative Languages:**

Declarative language programs specify relations or functions. The emphasis is on what is to be done, rather than on how to do it. An algorithmic engine embedded in the language implementation produces a solution based on the constraints specified by the programmer. The declarative language class may be split into two sub-classes:

A. **Logic Languages:** Logic languages are based on a subset of predicate calculus. A logic program consists of a series of rules and facts and a theorem to be proved. An example is PROLOG.
B. *Functional Languages*: Functional languages operate only through functions, which return a single value. The value of a function is determined solely by the values of its arguments and there are no state changes. An example is LISP.

Figure 1 graphically displays the relationships between languages.

![Programming Paradigms Diagram]

The above classifications are not mutually exclusive. In other words, a language may have the attributes of more than one paradigm. For example, ADA is a procedural language which supports some object-oriented features.

The discussion of which language to teach has now taken on a different tone. Authors of recent publications are arguing in favour of languages because of the paradigm that they represent, rather than because of features specific to the languages per se. As well, research into the learning of new paradigms is being undertaken (Carey and Shepherd, 1988). Many post-secondary institutes have followed M.I.T.'s lead in adopting a functional language (Scheme) as the first programming language (Abelson and Sussman, 1985; Reid, 1992). Object-oriented
languages are being taught from primary grades (Borne and Girardot, 1991) to university (Skublics and White, 1991; Temte, 1991).

Educators are finding that students schooled in one programming paradigm have difficulties shifting to a new one (Carey and Shepherd, 1988; Skublics and White, 1991; Wells and Kurtz, 1989). Therefore, there are some proposals to teach multiple paradigms in the first programming course (Appleby, 1991; Sebesta, 1989; Wells and Kurtz, 1989).

The maturation of the fields of computer science and computer science education have led to a cleaner differentiation of programming and programming languages. It would appear that a programmer working in one paradigm will have a very different view of programming than a programmer working in another paradigm. The paradigm chosen for a specific course may depend on the purpose for which programming is being taught. Furthermore, it is possible that different variables may predict success in different paradigms. Educational methodologies will certainly be affected by choice of programming paradigm. Thus studies of these current issues must be informed by an investigation of programming paradigm.
Research Perspective

Jacob (1988), in an overview of qualitative research in education, points out that within the last ten years researchers have been challenged to "transcend the limits of positivism" (p. 16). Magoon (1977), Wilson (1977), Howe and Eisenhart (1990) and Lincoln and Guba (1985) are among the scholars who have tried to make sense of what Howe and Eisenhart (1990) call "the proliferation of qualitative methods in educational research" (p. 2). Magoon (1977) contends that

the prime phenomena for the educational researcher may well be at their most basic level unavoidably sophisticated and highly organized. ... it is not surprising that a social anthropological methodology, which seeks to explicate the relationships among such phenomena as well as other factors, has such appeal (p. 652).

Although the term 'qualitative research' necessarily means different things to different people, Jacob (1988) says, four themes emerge from the discussion of alternatives to the traditional positivist approach: "the importance of conducting research in a natural setting", "the importance of understanding participants' perspectives", the necessity that "researchers subjectively and empathetically" know the participants' perspectives and the emergence of theories and questions from the data collection (p. 16).

Qualitative research is carried out in a natural setting and, as Lincoln and Guba (1985) state, "the major and sometimes only data collection instrument utilized in naturalistic inquiry is the inquirer him- or herself" (p. 267). Wilson (1977) says that "a researcher seeking to understand behavior must find ways to learn the manifest and latent meanings for the participants ..." (p. 253). Qualitative studies typically attempt to uncover these meanings through the use of interviews (e.g., Posner et al, 1982; Cardwell, 1988; Spradley, 1979) and case studies (e.g., Nussbaum and Novick, 1982; Yackel et al., 1990).
According to Lincoln and Guba (1985), “Events or situations are theoretically open to as many constructions as there are persons engaged in them, or as many reconstructions by a single individual as imaginations allows” (p. 77). They state that the researcher working in the “naturalist paradigm” (p. 37)

is likely to eschew random or representative sampling in favor of purposive or theoretical sampling because he or she thereby increases ... the likelihood that the full array of multiple realities will be uncovered; and because purposive sampling can be pursued in ways that will maximize the investigator’s ability to devise grounded theory that takes adequate account of local conditions, local mutual shapings, and local values ... (p. 40).

Studies of teachers’ constructions are based on the sense that “the practical wisdom of competent teachers remains a largely untapped source of insights for the improvement of teaching” (Feiman-Nemser and Floden, 1986, p. 505). “The prevailing view among most researchers is that teachers have experience while academics have knowledge” (Feiman-Nemser and Floden, 1986, p. 512). Various researchers have attempted to tap this source of useful experience (Elbaz, 1983; Cardwell, 1988; Lortie, 1975; Connelly and Clandidin, 1985).

Wilson (1977) states, in the context of case study research, that “participants must come to trust and value the observer enough to be willing to share intimate thoughts with him and answer his endless questions” (p 254). In the same vein, Woods (1986) discusses the need in an interview for a sense of trust, curiosity and naturalness on the part of the researcher. In his definition of a depth interview, the interviewer is viewed as a peer. The interview is open-ended and the respondent may ask questions of the interviewer. Spector (1984) also favours the open-ended interview. Massarik (1981) presents six interview typologies ranging from the hostile interview, in which “the atmosphere is one of mutual distrust and antagonism” (p. 201) to the phenomenal interview, which is “characterized by
maximal mutuality of trust, attaining genuine and deeply experienced caring between interviewer and interviewee and a commitment to joint search for shared understanding” (p. 203). Lincoln and Guba (1985) state that “the interview as utilized in naturalistic inquiry ... is usually unstructured” and “is almost always fully overt” (p. 269). The sense of a naturalistic interview, then, is that “[interviewer] and respondents, through repeated reformulations of questions and responses, [will] strive to arrive together at meanings that both can understand. The relevance and appropriateness of questions and responses [will emerge] through and [be] realized in the discourse itself” (Mishler, 1986).

As Spradley (1979) puts it, respondents are “nonanalytic” (p. 52) about their techniques; they respond “from the perspective of the insider” (p. 52) rather than from an outside theoretical perspective. In order to obtain the trust and openness of the respondents and in order to “subjectively and empathetically” (Jacob, 1988, p. 16) know their perspectives, it is useful for the researcher to be and be perceived in the interview situation to be a peer of the respondents. However, it is also useful for the researcher to express ignorance (Spradley, 1979) and to “constantly monitor and test his reactions” (Wilson, 1977, p. 259) so as to maintain the “tension in point of view -- between outsider and insider ...” (Wilson, 1977, p. 259).

We confuse ourselves if we believe that the people whose behavior we are concerned with, whether we observe them or interview them, can themselves provide an adequate explanation of their own behavior. That is our job, and the participant observer makes inferences from the data he collects just as the survey analyst makes inferences from the data collected for him (Trow, 1970, p. 148).

It is the job of the researcher to make sense, within his or her own framework, of the data collected. Eisner (1985) states that in order to “distinguish the significant from the trivial” the researcher needs “a set of ideas, theories, or models” (p. 221).
"We do not see and then assess significance" (p. 222), he says. "The essence of perception is that it is selective; there is no value-free mode of seeing" (p. 222). Spector (1984) cautions the researcher to "be on guard with regard to his own colors of observation" (p 461), but Glaser and Strauss (1970) comfort the nervous researcher by suggesting that she trust in her own knowledge when analysing qualitative data. "Categories and properties," they claim, "have a life apart from the evidence that gave rise to them" (Glaser and Strauss, 1967, p. 36). These categories are established not a priori, but as the data is gathered and as hypotheses are generated (Spector, 1984). Thus, to Spector (1984) (as to Lincoln and Guba (1985) and Glaser and Strauss (1967)) qualitative research involves not hypothesis testing, but hypothesis generation. According to her, descriptions of the world the subjects live in as they see it constitute a major segment of the reported results of a qualitative study. It is necessary, she says, to retain the subjects' own words because they provide "important insights into how they define their world" (p. 460).

Lincoln and Guba (1985) and Glaser and Strauss (1967) present slightly different versions of the constant comparative method of qualitative data analysis (Grove, 1988). The constant comparative method involves the identification in interview transcripts of individual data items (called "incidents" by Glaser and Strauss and "units" by Lincoln and Guba). These items are coded on index cards and then categorized intuitively by comparing individual items to one another. As the number of items in a category increases, the category is labelled and provisional rules are established which determine the items that are to be included in it. As overlapping categories are integrated, the number of categories diminishes. As Spector (1984) says, the constant comparative method is not meant to establish universality or proof of suggested causes but rather to provide category groupings,
identification of category properties and clues to the relationships between categories.
CHAPTER THREE

RESEARCH METHOD

This study examines the articulated perceptions of computer programming teachers in order to address the following research questions:

1. How do teachers organize their knowledge about programming?
2. How do teachers organize their knowledge about teaching programming?
3. How do teachers' constructions about programming integrate with their constructions about teaching programming?

Data Collection Procedures

Data consists of:

- two interviews with each of six post-secondary programming teachers
- copies of course materials (outlines, handouts) from each teacher

All interviews but one took place at the respondents' schools. The goal of the in-depth unstructured interviews was to elicit the respondents' views about programming and teaching. Thus, the interview protocol was used as a guideline, but interview questions were paraphrased and their sequence rearranged during each interview based on the respondent's replies. The section entitled Results of the Pilot Study, below, includes an annotated version of the interview protocol and a sample transcript is included in Appendix B. Before the interview sessions, the respondents sent me copies of a course outline and assignment from a language...
course they were teaching. I analysed these materials and prepared specific questions about their content. I also took to the interviews a program written in a language that the respondent was teaching so that it could be referred to as necessary (See annotated interview protocol below for details). The second interview was conducted within one week of the first and was used partially as an opportunity for both researcher and respondent to clarify and comment on issues arising from the first interview. After the first interview all respondents mentioned issues which they had been considering.

**Sampling**

"Purposive sampling" was utilized in order to uncover the "multiple realities" and "mutual shapings" (Lincoln and Guba, 1985, p. 40) held by teachers of each of the four programming paradigms outlined in chapter two.

Recall Figure 1 from chapter two, which classifies emerging programming language paradigms:

As described earlier, imperative languages deal with how actions are to be carried out. Imperative languages may be classified as either procedural (i.e., languages
which are based on the notion of functional units) or object-oriented (i.e., languages which are based on the notion of interacting objects). Declarative languages deal with what is; they specify relationships and depend on a built-in facility to determine how to produce a solution. Declarative languages may be classified as either logic (i.e., languages which allow the specification of rules and facts) or functional (i.e., languages which allow the specification of mathematical functions).

The sense that "a programming language ... [exerts] an influence on our thinking habits" (Dijkstra in Rich, 1984, p. 7) shaped the selection of respondents. The six people interviewed were selected because of the language paradigms in which they work. Table 1 outlines the paradigms with which each respondent is familiar.

Most respondents have had experience in more than one language paradigm and all have extensive experience with procedural languages. Claire’s experience is restricted to the procedural paradigm with minor exposure to a functional language. All of the other respondents have taught or used more than one paradigm and therefore were able to offer insights into the differences and similarities they perceive in them.
Table 1
Respondents' Paradigms

<table>
<thead>
<tr>
<th>Respondent</th>
<th>Procedural</th>
<th>Object-oriented</th>
<th>Functional</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Claire</td>
<td>Y</td>
<td>N</td>
<td>S</td>
<td>N</td>
</tr>
<tr>
<td>Isaac</td>
<td>Y</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Lorne</td>
<td>Y</td>
<td>Y</td>
<td>S</td>
<td>Y</td>
</tr>
<tr>
<td>Kyle</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Mario</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Ross</td>
<td>Y</td>
<td>S</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Note. Y = Yes. The respondent has taught or is currently working in the paradigm. 
N = No. The respondent has never taught or worked in the paradigm. 
S = Some. The respondent has at some point used the paradigm, but is not currently working in it.

Because this study focuses on the teaching of professional programmers, respondents were selected from two post-secondary institutes, the University of British Columbia and the British Columbia Institute of Technology. Neither institute requires that instructors have formal teacher training. Thus the respondents' instructional approaches and methods have been shaped by something other than exposure to formal teaching methodologies.
I conducted all of the interviews. Because I am a colleague and friend of the respondents in the study, the interviews had an openness that might not otherwise have been possible and allowed them to take the form of what Massarik (1981) terms "depth interviews" (p. 203).

**Interpretation and Analysis**

The interviews were audiotaped, transcribed and then edited for accuracy against the master tapes. I read each respondent's transcripts several times in order to get a sense of the respondent's use of language and metaphor and to extract salient items, representing "the smallest pieces of meaningful information" (Grove, 1988). Using the constant comparative method, I then identified themes which emerged from each respondent's interview sessions. These themes were grouped into two sections: the respondent's knowledge about programming, and the respondent's knowledge about teaching programming. The two groups were then compared for common themes and patterns. The results of each respondent's interviews were recorded in three sections: knowledge about programming; knowledge about teaching programming; and the integration of knowledge. In the final stage of analysis, themes and patterns were analysed across respondents, again using the constant comparative method. Differences and similarities between respondents were noted and more global themes and patterns were identified.

**Pilot Study**

A pilot study was conducted with two teachers from one of the institutes used in the main study. In this section I report the results of the pilot study interviews and explain the ways in which the main study was altered based on the experience I gained from conducting the pilot study. Both of the teachers interviewed in the pilot study, James and Melissa, have strong procedural backgrounds. Melissa's
experience is restricted to the procedural paradigm. James also has experience using the object-oriented paradigm.

James

producing really fascinating images

I have known James for ten years. He was my teacher for one year and I have worked with him for the past six years. Our interview took place in my office at the University of British Columbia, where James had come to hear a lecture about graphics. James referred to the lecture frequently during our discussion. As well, James referred to our relationship throughout the interview, using phrases like as you know, joking about my questions and referring to shared experiences.

James has taught computing at the same school for 18 years. During that time, he has taught decision systems and introductory computing courses to students not specializing in computing. The languages he has taught include BASIC, FORTRAN, APL, C and C++. The first four of these languages are procedural. APL shares some of the features of a functional language. C++ is a hybrid procedural/object-oriented language. Currently James is teaching decision systems and an advanced course in computer graphics. In both courses C and C++ are being used. Before being hired for his current job, James had no teaching experience. Neither did he receive any teacher training.

James’ first exposure to computing was in the 1950s when he was working on applications involving the calculation of trajectories for guided missiles. He did not learn to program until 1963, however, when he taught himself FORTRAN in order to solve structural analysis problems. James’ industry experience has been in the areas of aeronautics and operations research. He has a PhD in operations research.

What Teachers Know About Programming
James made use of many visual metaphors during our interview. He used phrases such as *I could imagine* and *the way I see it* frequently. He said that *we've put blinkers on* the students by teaching them a certain style of programming. James also employed many physical images in his speech. He said that he was *bowled over* by the use of computers, that he *fell into* teaching and several times expressed the opinion that programmers *wrestle with* their problems. James told several stories about his experiences in industry and in the classroom and often used very detailed examples to illustrate his points. He was able to remember specific details about data structures used in a program he wrote many years ago.

**James' Knowledge about Programming**

Programming to James is a means to an end. To him, programming is a tool which allows him to *tap into the power* of a computer. This power allows him to *save time and effort* when *producing fascinating images*. James is intrigued by the power and results produced by computers. He said, *All of that work involved in number crunching, I could actually get it done by ... writing a few lines of code .... That fascinated me.* James finds computing *fun* and *interesting* and says that it is intriguing to *program away and then suddenly see some interesting results on the screen*. He enjoys the feeling of *power over that machine*.

James organizes languages by application and power. That is, he separates the languages he knows, which are all used for scientific applications, from languages used for business, like PL/I, or for list processing and logic, like LISP and PROLOG. C and C++ for him are a subset of the scientific languages. He sees them as modern, more powerful versions of the other scientific languages and he likes them more than he likes the others.
James is attracted to languages that are efficient, powerful, easy to use and readable. When he learned C, he realized that it was [his] language -- that was the language that [he] liked. Now he says he is realizing that C++ is the way to go. James is attracted to C++ because it fits the way he thinks. He says that it models the real world for him. It seems to be like the real world, where there are objects and these objects do react with an environment. In general James feels that we tend to choose the language that matches the way we think.

James sees programming and analysis as very different things. Both are concerned with solving a problem, but whereas programming is detail-oriented and just a way of producing a picture, analysis is a more conceptual activity which involves thinking and modeling and theory. To James, the programming ... is not so important as the models.

James’ strong distinction between programming and analysis is echoed in his view that there are two different kinds of programmers. Business programmers are concerned solely with programming. They are working continually with programs and doing maintenance. Modeling for them is trivial. Scientific programmers, on the other hand, are using scientific applications to do analysis. For them, programming is secondary to modeling. They’re not continually at the screen. They’re spending a lot of time looking at the data, looking at the models and then, of course, programming.

**James’ Knowledge about Teaching Programming**

James fell into teaching. He at first refused the offer of a teaching job, because he didn’t like the idea and ... hadn’t even thought about doing it. It was difficult for him to adjust to teaching, because he found that although he thought [he] knew the [course material] ... [he] realized [he] didn’t. Unfortunately [he] sort of experimented
James likes the flexibility of teaching at his school and finds it amazing that we're the ones that are doing the learning. He likes the fact that he is learning all the time and says, If I weren't learning I would leave. James dislikes the pressure he feels in his job, but finds that he learns with that pressure. As he puts it, I hate the pressure, but I like the learning that comes along with it. To James a good class is one during which the students are interested and ask good questions. However, he does not like being unable to answer students' questions. He feels that being a teacher ... [he] should know the answers.

James feels that programming should be learned as a means to an end. He doesn't actually teach the details of programming, but his students learn and use programming techniques in order to solve application problems. When asked about the value of teaching programming in elementary schools, James responded, It's a great idea. He thinks that programming can make learning fun and interesting. He says that he found his own school days totally boring, but thinks that students using computers could teach themselves a lot of the stuff ... and enjoy doing it rather than being bored and watching. James feels that understanding comes from doing, in both programming, where one understands functions better if one designs them oneself, and mathematics, where one understands calculations better if one does them oneself.

James' preferred mode of teaching is a combination of lecture and lab, in which he introduces a concept and then has students implement it. Depending on what
happens [in the lab] he and the students would then try something new. James uses a spiral approach to teaching programming. That is, he works through the concepts, moving from data types to structures, showing example programs to the students and of course getting them to write their own programs. As James says, I tend to teach applications and these applications require programming and obviously a part of my work involves helping students with debugging and such and occasionally talking about approaches and algorithms, but I don’t actually teach the details of programming.

In order to help students achieve deeper understanding, James attempts to show them various views of a problem. His image of a perfect teaching environment is one in which many different views are available to the students. His utopian classroom would include networked UNIX workstations so that students could see the problems in many different ways easily and so that he could send out pop-up windows to the students. His classroom would also have a projector to allow students to see what is on his screen, video projectors, an overhead projector and a whiteboard.

James looks at specific programs first from a general perspective and then in terms of specific details. When asked to explain a piece of code or to describe a program, he first gave an overview of functionality and then moved on to explain specific details, mentioning particularly challenging aspects of the code. When asked to describe how he would help a student understand a piece of C code which loops through two floating point arrays assigning one to the other, he first made certain that the ‘student’ understood the basic concept of an array. He then drew a picture of the array and gave an explanation in broad terms without going into details. This broad explanation afforded some insight into the mental models James employs when explaining program code. The array, to him, is a column of numbers. Each

*What Teachers Know About Programming*
element in the array is a box. James assigned a value to each element of the array just to make it more .... The values were written with one decimal point of significance. James' tendency to look at programs from two points of view, the general and the specific, is also apparent in his description of the traits of a good programmer.

Debbe: What are the attributes of a good programmer? ...

James: Well, they have to be able to think really well to be a good programmer. There are some people who ... program really well, but they don’t ... really know what they’re doing because they ... can produce beautiful code, but ... they need, I would imagine, a lot of guidance. ... they’ll ... start programming and then they’ll continually want to know what to do, while I would say a good programmer -- my version of a good programmer -- would be somebody who can think and at the same time program well. But they’re able to think and see the big picture.

Programming, he said, requires a strange mix ... of someone who thinks broadly and at the same time can implement those broad thoughts through the details of a program.

James sees a major distinction between procedural languages and object-oriented languages. Students at his school, he says, just can’t handle Smalltalk.

Debbe: Why do you think that is, that our students can’t handle Smalltalk?

James: Well, I think it’s because they’re so stuck with this older, not this older, but this other way of doing things that’s separation between function: program and data. And that the idea of working with objects is strange to them.

Debbe: So it’s a matter of their previous experience?

James: I think so. Yes, I think we’ve drilled them so heavily into this idea of data and programs, functions requiring data, returning something, that we’ve put blinkers on them.
An Integration of James’ Knowledge

Several patterns link James constructions about programming to his constructions about teaching programming.

Most striking is the strong sense of the computer as an aid to visualization and modeling. James’ interest in visual imagery and models is apparent in his speech and in his choice of graphics and decision systems as subject areas. His attraction to computing stems from his ability to use the power of computers to create fascinating visual models. In our discussion, when he described the results of his programming efforts, he talked about colouring pixels on the screen, producing images, and seeing what came out. He feels that students need to see various models in order to learn. That is, various forms of media need to be used as teaching aids and students need to be shown conceptual models in order to understand specific programs. James’ feels that good programmers must be able to model the world. His explanation of why students have trouble with object-oriented programming epitomizes James’ concept of teaching as presenting various views: teachers put blinkers on students by teaching them only one paradigm.

Also important to James is a sense of learning by doing. He taught himself programming and, by trial and error, learned how to teach. His sense that one understands best by doing is manifest in his view that programming is best learned by practice and his view that learning can be made meaningful by having students learn how to program solutions. James loves learning new things, so he loves teaching in a new area in which he and his students can together make new discoveries. Good programmers, James feels, do not need too much direction. They should be able to think well on their own.
James' view that computing involves the ability to see the big picture as well as the ability to deal with specific details is apparent in the strong differentiation he makes between programming and analysis. In addition, it allows him to distinguish business programmers from scientific programmers. The former are preoccupied with programming, while the latter spend their time on analysis. This firm distinction between analysis and programming shows up in James' explanations of code segments, as he first gives a broad view and then goes into detail, and in his conception of what makes a good programmer, which includes the strange mix of high-level analysis and low-level detail work.

James' conception of the traits required to be a good programmer touches on all of the themes mentioned here: a good programmer can see the big picture as well as the detail, does not just write code, but applies knowledge to an application area, can model the world and works well independently.

For James, programming is a means to an end. He learned programming in order to use the computer's power to perform scientific applications. He selects languages based on their power and ability to model the real world. Programming is just a way to produce a picture. His role as an instructor is that of someone who introduces concepts which the students then implement. The fact that the students are programming in order to achieve their goals is secondary to the end product, which is usually a picture. His sense that programming could be used to make learning fun and meaningful further supports his conception that programming is simply a tool with which to produce really fascinating images.
Melissa

*a sense of mastery*

I have known Melissa for ten years. She was my teacher for part of one term and I have worked with her for the past six years. We have a personal friendship in addition to our collegial relationship. Melissa’s two interviews took place one week apart and were conducted in our school office after her classes were finished for the day. The interview discussions felt very personal. Melissa referred often to situations we had discussed outside of the interview setting. Several times during the interview she asked my opinion about a question I had asked her.

Melissa has taught computing at the same school for 11 years. During that time, she has taught BASIC, Pascal, FOCUS, RPG, Fourth Generation Languages (4GLs) and introductory computing courses to students not specializing in computing. This year she is teaching COBOL, PL/I and assembly language. All of the languages that she knows are procedural. Before being hired for her current job, Melissa taught a night school introductory computing course. She has also taught motorcycle riding and is currently a flying instructor.

Melissa’s first exposure to computing was twenty years ago at the age of sixteen. Her brother was taking a BASIC programming course at university. She became interested in programming and went through the course on her own. Melissa took two years of math and science at university feeling frustrated because she was not able to specialize in computing. She therefore enrolled in the school at which she now teaches, where she was able to specialize.

Melissa worked in programming and support in minicomputer environments for four years before returning to her alma mater as a teacher.
Melissa’s speech is striking in its use of physical metaphor. Her interviews were peppered with phrases such as *take a step backward, grab onto it, tie your hands behind your back* and *I’ve got to dig down*. As well, Melissa used many hand motions to illustrate her physical descriptions. The stories she told were about particular students and classes she has taught or is teaching.

**Melissa’s Knowledge about Programming**

Like flying and motorcycle riding, programming gives Melissa a sense of mastery and control of machinery. I asked her to analyse the way she thinks about programming:

Debbe: What do you think affects and build the way you think about programming?

Melissa: I like to be in control and master things. ... It’s the same with flying, though. I want to take this piece of equipment and put it through its paces and do the perfect landing and the perfect turns and make it do what I want it to do. I’d say that would be the most important concept is a sense of mastery.

To her, programming is a big puzzle that you [get] to tear apart and put back together again. Programming caught [her] imagination at an early age because it was not mainstream ... and [she] always liked things that you could tinker with and mind games and mathematics and puzzles and things. She still likes the sense of being in control, of mastering a problem with a certain degree of finesse which was the initial thing that attracted her to programming. For Melissa, what is important is not just solving the problem, it’s how you get there.

To Melissa, programming means breaking a large problem down into manageable pieces that can be solved. Programming is a very tangible activity for her. Melissa learned assembly language in her first term at school. She said, *I found it horribly*
complicated and ... struggled and didn't do nearly as well as I thought I should be doing in first term, but once that fell into place I never really found that I had that many problems learning another language. Melissa feels that because she learned assembly language in her first formal course, she constantly picture[s] what's going on in the background in any language she now uses. She actually sort of picture[s] things physically happening as a result of [a program instruction]. She sees a honeycomb of storage boxes and what is contained in them ... and ... think[s] three dimensionally in [her] head. Her vision is very detailed. She sees data going into the Arithmetic Logic Unit, the calculation being done and a result being placed somewhere. Melissa pictures languages in different ways. With PL/I, she thinks in blocks, whereas in assembly language she thinks more in a straight line where you go out and back and out and back. During the interview, Melissa used distinct hand motions while describing things such as getting a PL/I building block to plug into her program.

Melissa has experience programming only in the procedural paradigm. She thinks that one procedural language for the most part is like another. They differ only on how much detail you have to think about. Her sense is that one might think differently in another paradigm, but because of her lack of non-procedural experience she can't offer a knowledgeable opinion.

Within the procedural paradigm, Melissa classifies languages in terms of how much she enjoys coding in them, which is determined by how awkward she perceives each language to be. Melissa most likes assembly languages because they give her a high degree of control over low-level machine functions. Her least favourite language is RPG, which is to be avoided at all costs because it is impossible to debug. She said, You're better to pitch it out, throw it out, rip it up and rewrite it than you were to try and debug it. COBOL and BASIC are unwieldy and awkward to debug.
and code. In particular, she disliked BASIC intensely because it wasn’t elegant. I asked what she meant by elegant.

You couldn’t do tricky things with it .... For example, you couldn’t go in and manipulate bits and and move bits around and ... it just seemed to tie your hands behind your back so much. Difficult to go through and do some trickier things which is what I used to like to do.

COBOL does not allow

in-line logic and it forces you down to deeper levels, deeper and deeper and deeper and it gets quite muddled when you’re trying to understand what’s going on. The language structure itself with sentences and paragraphs and there’s a period at the end of the whole thing. I find the structure itself becomes very cumbersome.

PL/I and 4GLs she finds easier to work with. PL/I, in particular, has nice built in structures and functionality.

When asked what image comes to mind when she hears the word ‘programmer’, Melissa said that she’s torn between the typical nerdy student with a calculator hanging off his belt and tape on the glasses and (pause) us! Her view of typical programmers is dictated very much by the students that she sees. Overall, though, she sees them as hard-working, detailed, able to handle the job stress ..., able to concentrate for long periods of time, ... able to ... see the overall picture and then switch it into being able to handle the detail.

**Melissa’s Knowledge about Teaching Programming**

Melissa teaches programming and flying and has in the past taught motorcycle riding. As she puts it, I teach everything I learn how to do. Melissa likes teaching because she likes being in control. She likes talking and having 150 people listening. She likes the feeling she gets when, after a class in which the students did not
understand what she was teaching, she goes home, thinks about it and comes back with a different angle and sees the lights go on.

Melissa feels that it is a good idea to teach programming in elementary schools in order to demystify computing; to give students the understanding that the computer is not HAL from 2001 running amok and doing things all on its own. It still is somebody's thought processes behind it all. Whereas learning application packages would be easy for students and would give them a degree of control and a sense of satisfaction, in order to demystify the process you have to actually get in and write a little program to ... understand what the machine is doing.

During our discussion, Melissa spoke extensively about students' reactions to various languages and her perceptions of changing reactions over the years. Several times she volunteered information about student attitudes and expressed the opinion that students need to experience success and gain a sense of confidence in order to learn programming.

Melissa feels that there may be a correlation between students' feelings about an instructor and their feelings about the language being taught. She tries hard to hide her biases against certain languages, but is eager to share her enthusiasm about others. Melissa's concern for the students is made clear by her feelings that although she wouldn't choose COBOL to do anything and despite the fact that teaching COBOL wouldn't be [her] first choice, given the skills of the faculty at her school, she is probably the best person to teach COBOL.

I probably do a better job of it and know it better than anyone else. So if they're going to be taught COBOL, I'd just as soon at least they be taught it by someone who knows it and I probably dislike it less than anyone else here.

She senses that the students have reacted positively to her enthusiasm about PL/I.
Melissa feels strongly that students will solve problems in the style of the language with which they are familiar. In particular, she sees her second year students writing PL/I programs the same way they would a COBOL program cuz COBOL has some restrictions. Having students submit preliminary design work allows her to correct their logic -- streamline it -- so that they write their first PL/I program in a very nice PL/Ish way. They lose the old COBOL restrictions. Melissa maintained that programmers think of a shell of a program in their minds. Depending on which languages one learns first, one will have access to different shells. Having written programs in several languages, she has a catalogue of shells in her mind and can pick out, for example, a PL/I shell if she needs one. If she is forced to code in a language for which she does not have a shell, she will look at sample programs to get a sense of the language. Students, she claimed, go back to the most recent programming language they have learned.

Melissa looks at programs in a very systematic sequential way. She explained program code to me line by line and, when asked to describe how she would help a student having difficulties, approached the problem in a very linear fashion:

So I would go through it with them and make sure there were no compile errors and then I'd say, well, it's interesting, notice here the //EXEC. It started to execute, but nothing came out. So let's trace it back. Where would it first print something out? And then I'd start going through line by line. If we still couldn't figure it out I'd say, well, it's really important that we figure out how far it got, so let's run it again with some display statements, display checkpoint one, checkpoint two, checkpoint three and run it again and see how far it got and that way we'll start to narrow down the range of where it didn't work.

Melissa paid a lot of attention to the details of the programs. She looked first at the data definitions and made a note on paper of the field names. She immediately found a spelling mistake in some sample code.
The structure of Melissa's courses parallels the way she looks at programs. She spends about one third of the term teaching data definitions in detail. She then goes through the logic processing constructs and spends the last part of the course dealing with issues arising from student concerns.

Melissa makes extensive use of mental models so that the students start to understand when things go wrong. She wants them to understand why they get a data exception. Melissa draws pictures trying to get them to visualize ... and [to be] able to see this as something they can actually see (she picks up a piece of paper) ... rather than something inside the black box that happens. She draws boxes to represent storage locations and shows their contents in hexadecimal and binary notation.

Melissa feels that the students' first language should be assembly language. In the past, when students learned assembly language in their first term, she found that they absorbed it and .... were able to use it in subsequent languages. She doesn't think that students who learn Pascal or C as a first language are able to picture what is going on in the background when they are programming. Several times since we completed our interview sessions Melissa has reiterated her view that we should go back to teaching assembly language in term one. She called me at home to read a note from a student explaining that he finally (after three terms of programming in other languages) came to understand programming when he studied assembly language. She said that this is more evidence that assembly language should be learned in term one.

Melissa feels strongly that certain people do not have an aptitude for programming, just as she does not have the aptitude to draw or play a musical instrument. Good
programmers, she feels, must have the desire to learn programming and must be able to think logically. Melissa described a friend of hers with whom

you cannot have a logical conversation .... Her mind just ... skips around things (she demonstrates with hand motions) and she seems to be pulling bits and pieces out of the air. ... And it's hard to get a straight progression of thought, cause and effect, the connections between things, but she paints beautifully and ... she can take a canvas and put a little blob here and a little blob there and after a while it starts to turn into something. And I'm not sure that she knows all along that that's what she's going to end up on, but she eventually turns it into that. She is a friend who I don't think I could ever teach her how to program. I honestly don't. ... because I don't think she'd want to and ... she couldn't sort of force her mind into a thought process that considered cause and effect a little more. She's a hairdresser. So she cuts a bit here and cuts a bit there and looks at it and says, oh, maybe, OK, I'll cut a little more here and I'll change it here, rather than sort of plotting the whole thing out.

In addition to being motivated and having a strong sense of cause and effect, then, Melissa feels that programmers need to be able to plan. During the interview, she told a story about a student who would plan her programs for only the ideal scenario rather than writing a program that could handle various cases. Now the student is plugging all the holes in the dam. Planning, to Melissa, involves being able to anticipate many outcomes. Melissa feels that good programmers need to be persistent. They need to be able to handle frustration calmly without becoming irritable. As well they need to be free of a sense of intimidation.

**An Integration of Melissa's Knowledge**

Programming gives Melissa the sense of control and power over a machine that she likes. To her, programming is a puzzle to be mastered and she is therefore attracted to languages that give her the power and control to solve problems with finesse. Assembly language lets her do tricky things at a low level. PL/I gives her power and flexibility. Cumbersome languages like BASIC and COBOL tie [her] hands behind
[her] back, [force her] down to deeper levels. In other words, the language, not Melissa, is in control. Melissa likes the sense of control she gets from teaching as well. She likes lecturing and feels that a teacher's biases can strongly affect students' attitudes. Melissa checks students design work so that she can control the shape of their programming logic. Melissa feels that students need a sense of control in order to be good programmers, so she attempts to give them power by building confidence. She maintains that computing could be demystified by giving students a degree of control, by showing them that computers are not in control, but humans are. Overcoming intimidation about machinery allows a feeling of control.

Melissa gains her sense of control over the machine by understanding all the details of what she is doing. Although she recognizes that programmers need to see both the big picture and the details, for her the important aspect is the ability to break large problems down into smaller parts. She categorizes the languages she knows in terms of the level of detail accessible and pays strict attention to the details of a program she is examining. Because she is so concerned with detail, she advocates the teaching of assembly language as the first language so that students can understand what is going on in the machine.

Melissa's understanding of what is going on in the machine, and the understanding she attempts to share with students, is a very tangible one. She sees the storage and the operations as she goes through a program. The mental models she employs when teaching reflect her very physical sense of what is happening when a program is executing.

For Melissa, programs operate in a linear fashion and programming is done in a very systematic way. Melissa feels that to be a good programmer one must think logically, have a strong grasp of cause and effect and be able to plan for any
eventuality. She reads and explains programs line by line and when helping students with problems, traces through their programs step by step. Her course structure echoes this linear approach to program design and execution.

Results of the Pilot Study

The pilot study gave me additional experience in interview techniques and accentuated the need to allow the respondents adequate time to reflect upon and respond to both individual questions and the interview as a whole. After the first pilot interview, the protocol was separated into two sections. The second respondent was interviewed twice over the period of one week. Based on the pilot study, the interview protocol was shortened and modified. As well, I decided to ask respondents for course materials before the interviews so that questions about course structure and content could be more concrete. During the main study, respondents' language knowledge was recorded on index cards to facilitate language classification. Analysis of the pilot transcripts allowed me to experiment with various techniques and to develop a procedure with which I felt comfortable. Because the interview protocol was not altered drastically after the pilot study, and because the pilot study elicited information about the same themes as the main study, the results of the pilot study were analysed together with the results of the main study.

Based on the pilot study, the interview protocol was modified. Table 2 contains an annotated version of the protocol used in the main study. As mentioned previously, the protocol was used as a guide. The actual interview questions varied depending on the respondents' replies. See Appendix B for a sample transcript.
Table 2

Interview Protocol

Introduction:

Purpose
Anonymity/Right to refuse
Sense of the interview "Rapport"

In keeping with the overt nature of the study, respondents were informed of the purpose and atmosphere of the interview.

Background:

1. How did you come to be teaching at your current school?
2. How long have you taught here?
3. What courses are you teaching now?
4. Tell me about past teaching experience. where/what/to whom/languages

This section was used to elicit information about the teaching experiences that have helped shape the respondents' views.

Professional Background:

1. Describe how you first learned to program. why/where/what environment?
2. Describe your professional/research experience.
3. What programming languages do you know/have you learned? (Put them on cards)

This section was used to elicit information about the programming experiences that have helped shape the respondents’ views.
Thoughts about Programming:

1. When did you last write a program? Describe it to me.

2. What do you like about programming?

3. What do you dislike?

Thoughts about Language:

1. You know the following languages: How would you categorize them? (Use cards to do this). Explain.

2. What is your favourite language? Why? Can you generalize - what attributes attract you to a language?

3. How do you choose the language you use to code specific programs? eg. a program to sort some data, a program to diagnose a problem in a circuit board, an Accounts Receivable module

4. Some people claim that the language we code in shapes the way we think. Others feel that we select languages based on the way we think. Please comment.

5. Does it feel different to code in X than to code in Y?

With this group of questions I hoped to learn about the ways in which the respondents think about programs and about their ideas of what programming entails by listening to the ways in which they described programs and programming.

Here I hoped to discover some sense of the ways in which the respondents organize their knowledge of language. I hoped to elicit information about how the respondents differentiate between languages and what aspects of languages are important to them.

What Teachers Know About Programming
Thoughts about Programmers:

1. What image comes to mind when I say the word "Programmer"?

2. What is that person doing?
   physically
   mentally

3. So programming means ...

4. Would you call yourself a programmer?

5. What are the attributes of a good programmer?

Thoughts about Teaching:

1. Tell me about a good teaching day

2. Tell me about a bad one

3. What do you like about teaching?

4. What do you dislike?

This section was designed to find out more about the respondents' views of programming. By asking about programmers, I hoped to discover not only the attributes that the respondents find important in programmers, but also the aspects of programming that are important to them.

These general questions about teaching were asked to discover the respondents' feelings about teaching in general. Questions three and four were asked only if the first two questions did not elicit general concepts.
Thoughts about Teaching Programming:

1. Programming is now taught in elementary school. Do you think that this is a good idea? why (not)?

2. What is the purpose of teaching programming at such a young age?

3. Do you see the same purpose to your teaching?

4. Question about course outline.

5. If you had infinite resources, would things be different?

6. Question about assignment.

7. Are you consciously trying to impart a particular image to the students?

8. Explain this code to me as if I were a student coming to you for help.

9. What’s your favourite language to teach? Why?

10. What’s your least favourite language to teach? Why?

11. (Why) Are there differences?

12. There is a lot of controversy about the best first language for students. Comments?
Thoughts about Students:

1. Can anyone learn to program? Why (not)?

2. Think of 3 people you know (not programmers/students). Would they be good programmers? Why (not)?

3. Some people claim that they can tell almost instantly whether or not a student is going to "make it". Can you do this? How? Are you usually right?

Conclusion:

1. Anything that has puzzled/interested you that you want to pursue?

2. Thanks

In this section I hoped to uncover information about the respondents' views of how to predict programming success. As well, the information gained here could be compared to the information elicited from the section asking about programmer's attributes. By asking about people other than programmers or students, I hoped to discover whether the respondents felt that they could predict success based on something other than students' behaviour in programming classes.
CHAPTER FOUR

RESPONDENTS' KNOWLEDGE ABOUT PROGRAMMING

The six respondents in the main study each have personal interpretations of programming and teaching programming. In this chapter, the results of my discussions with each respondent are presented. Each teacher's educational and employment background is outlined, my relationship to the respondent is described and a sense of the tone of the interview and the respondent's manner of speaking is given. I then present summaries of the respondent's articulated knowledge about programming and about teaching programming. The section for each respondent concludes with an analysis of the ways in which knowledge about programming integrate with knowledge about teaching programming.
Ross

dancing carrots on the screen

Ross has taught programming at the same school for 17 years. At the beginning of his teaching career he taught procedural languages (assembly language, COBOL, Pascal) and he still teaches courses in procedural languages from time to time. For the past eight years, however, he has been the head of the expert systems area at his school and in recent years has taught almost exclusively in the area of artificial intelligence using LISP (a functional language) and, to a lesser degree, PROLOG (a declarative language). Before beginning his teaching career, Ross worked in a mainframe environment, doing systems programming in assembly language and working with COBOL. He was a programmer, supervisor and manager in the mainframe environment for approximately ten years.

During our interview sessions, Ross and I focused primarily on the differences between programming paradigms. The sessions felt very personal. In the past few years, Ross and I have often discussed programming paradigms and in the course of the interviews Ross frequently alluded to our previous conversations. This sense of shared understanding led him to sometimes reject superficial, standard responses in a search for deeper understandings. We joked and laughed a lot during the discussions and despite the fact that Ross felt that he had been doing all the talking, to me our conversation felt similar to our usual discussions. Ross appeared to want to be able to help analyse the responses he gave me. Several times he answered my questions by saying that he did not have a list to offer. He seemed to be saying that although he could help with specific examples, he had not done a thorough analysis and so could not offer a complete response.
During the interviews Ross used very sensory language. He spoke as though he physically inhabits the world of a programming paradigm. He said, for example, *The world that I spend most of my time in these days is a sort of LISP-world.* Ross used many physical metaphors. He talked about *stumbling on* ideas, *plucking* them out of the air, *wallowing* in a problem and, as a programmer, *getting inside the box.* In addition to sensory physical language, Ross used three metaphors extensively: he talked about labels (*labels of categories aren’t coming to my head, for labelling purposes let’s talk about ...*), patterns (*the pattern matcher in me, techniques, patterns, algorithms that he’s got*) and lists (*I don’t know if I’ve got a list ready, make a great big list of all the utilities*).

**Ross’s Knowledge about Programming**

Ross has a strong sense of the various paradigms extant in the programming world. He spoke about a cluster of paradigms and pseudo-paradigms with overlapping and non-overlapping attributes and illustrated his framework by drawing a Venn diagram in the air. His language and comments suggested that he feels as though one actually inhabits the world of a programming paradigm. We spent the majority of our time discussing the distinctions he sees between two worlds in particular: the world of block structured languages and *LISP-world.* He seems to differentiate between them in terms of the amount of freedom and control they offer the programmer; in terms of the amount of discipline they impose.

Ross used to live in the block structured world. He worked as a programmer for several years and then became a supervisor in that same world and then a manager in that same world and so on and so ... your fingers stop doing the programming, but you’re still, you know, the world is still the same world. ... and then teaching here, there was only the one world. Ross spoke at length about COBOL, and
explained that COBOL was his shorthand for block structured language ... old-fashioned block structured language at its worst ... generic. Block structured world, to Ross, is a world of flowcharts, design tools and desk checking, a world where the programmer does extensive thinking and designing before beginning to code. It is a world of tight specifications, formal authoritarian languages and cut and dried problems. As he put it, COBOL is always just COBOL. Any particular session with COBOL feels like any other session with COBOL. No real surprises. A certain amount of plodding and you get where you're going. No big breakthroughs. Nothing wonderful ever happens, nothing horrible ever happens in COBOL.

Ross, in a slightly mocking tone, said that I might find interesting his path to enlightenment from the block structured world. His first exposure to LISP was in a course where the professor taught concepts of artificial intelligence, but did no language teaching. Ross's early LISP looked like COBOL or something, felt like COBOL. No, it felt worse than COBOL. It felt very frustrating. Ross remembers using design methodologies that had spilled over from the other paradigm, remembers physically needing to look at a hard copy of his design work. Through reading and talking to others and just kind of stumbling on it though, gradually it dawned on Ross that LISP was an interactive language with a capital I on interactive.

Now, he said, he never [goes] anywhere near paper. He rarely goes near a manual and finds forgotten syntax by trial and error. He said,

To caricature it a bit, you get the barest inkling of an idea and you try it out right there and then on the machine .... It's, have a thought, try it out, have a thought try it out .... The general notion ... is sort of the machine as the scratch pad, the machine as your cocktail napkin.
Ross said that aspects of LISP make programmers start thinking differently about what they are doing. This different way of thinking makes one feel *much less a slave to the tool* that one is using.

LISP gives Ross a feeling of control at the level at which he is comfortable. That is, he is not interested in having control at the register or pointer level, as one has with assembly language, but wants to have control at the inference engine level. He wants to know that he can *get inside the box*, that there is *no mystery*. Ross said that Smalltalk and PROLOG have some of the same advantages as LISP. However, Ross dislikes both languages. Neither give him as much control as he has in LISP; he can't get at their *inner workings*. In Smalltalk this means that he is forced to use objects that behave in a *generic way* as opposed to the way he wants them to behave. In PROLOG he is unable to customize his own inference engine. He said that he is *not a PROLOG fan*. PROLOG offers so much and goes through periods of delivering beautifully and then lets you down and lets you down big. In particular, he feels that when debugging or optimizing performance, one is often forced to understand what is going on procedurally in the background of a declarative language.

Ross talked about having had the same sense or spirit with systems programming as he has with LISP. He told a story about a problem he worked on as a systems programmer where he *stumbled on* a phenomenon and *started to experiment*. As he said, *I just kind of wallowed around in it and eventually through trial and error and luck, ... , found something that was quite good*. LISP, he said, is known as the *machine language of AI* and although *assembly language and LISP are about as far apart as you can get ... somehow it's ... the same activity.*

Ross said that LISP is good for *feeling your way into unknown territory*, which is what one is doing with *AI (artificial intelligence) flavoured tasks*. Whereas other
languages are what they are, Ross stressed the notion that LISP is a chameleon. It can be made to mimic other languages. Any features that Ross covets in another language, he can steal and write in LISP because of its chameleon-like nature. Any time you write a LISP program, one way to think of it is you're writing a language that perfectly matches the domain, the application you're working on. LISP modules are tiny, so you can always think about any bite-sized chunk with a good rich deep understanding .... And the granularity is just, just perfectly matches your own need for granularity. Why? Cuz you set it up that way.

Programming to Ross is a high. The high used to come from writing code, but as he moved along the industry promotional career path he found that one could still do all the creative stuff and all you're giving up is the keying or the moving of the pencil. So all the design stuff and so on is still yours .... For him, the high moved from writing the code to thinking up the solutions. I asked him where the high came from.

Debbe: So that programming high comes from creativity?

Ross: I think so. It's hard to say what that means, but it's something to do with discovering a pattern in the real world or inventing one, it doesn't matter, and finding a way to turn it into something in the machine. ... Well, here's one example. There was a student group last term. ... The application was going to be gardening based. We knew that for sure, but we didn't quite know who to target it at so anyway [I said], "we could have a this or a this or we could have dancing carrots on the screen" and the high came from the fact that ten weeks later we had dancing carrots on the screen. So part of the high was the actual pulling it off, but it was the notion that you could make something up out of the air -- anything, anything (we laugh) -- and make it happen. You know, we talk about creating one's own reality and so on, but it's like you can!
Ross said that he wishes he had the talent to sculpt or draw, but thinks that perhaps he's found the medium that allows him to convert stuff from imagination to a kind of reality.

Ross uses LISP for any programming he now does, even the most non-AI thing that you can think of. LISP is his language of choice. It seems natural to him. He recognizes, though, that other languages seem to fit other people. For example, There are people in the world, he said, whom I respect, who love this thing, PROLOG. And build things out of the air with it with apparent ease. Of a former fellow graduate student he said, PROLOG fit him and LISP fit me for reasons I don't understand.

Others seem to find a fit in the block structured language world that Ross used to inhabit.

Ross: ... there are programmers around who make a really nice living out of applying fairly standard formula kinds of approaches to stuff in the world ...

Debbe: Could you ... say that someone who does that type of programming ... is a good programmer?

Ross: Not in my world, but in worlds that I've inhabited, yes, for sure, absolutely.

Ross mentioned that he sometimes makes a joke about playing off one way of life against the other.

You can caricature either ... so [LISP-world programming] at its worst is undisciplined hacking, juvenile behaviour ... and the other is mindless drudgery .... And if you want to hype them both then the one is inspired creativity and the other is ... keeping the engines of society oiled and working as they should ... doing the data processing tasks of society and I choose that phrase deliberately. The people that do well in the one group I think are a very different kind of critter from the people who do well in the other group.
Ross said that he sometimes categorizes languages as LISP and other. At the end of the second interview, as he looked at the half dozen index cards upon which I had written the names of languages that he knows scattered on the table, he said, 

*physically I want to push these things (other than LISP) away.*

Ross: It’s not so much that I’m wild about [LISP]. I’m wild about something that doesn’t exist yet. But I have at least the illusion that I can build it out of this.

Debbe: Mm-hmm. What is that thing that doesn’t exist yet?

Ross: Well, it doesn’t have a lot of parentheses in it for one thing. (we laugh) And it’s got ... a very smart interface that doesn’t make me do the grunt work the way all these things do. And it doesn’t require compiling and what? It doesn’t get in my way. It lets me worry about my application. It lets me build dancing carrots out of the air one minute and something very very different out of the air the next minute ...

### Ross’s Knowledge about Teaching Programming

Ross’s course outline states, “LISP is VERY different from the languages most students know, and the course uses it as a vehicle to introduce students to symbolic rather than numeric uses of computers.” Ross said that he [drags the students] kicking and screaming away from their flowcharts and various design tools. After having learned LISP for one term, Ross’s students, along with others who have not learned LISP, study Smalltalk. I asked him about his and other students’ reactions to the course. [My students] have certainly seen how LISP can look like an object-oriented language ... and therefore when they get a Smalltalk course ... they make a nice direct match of one to the other. His students tell him that they are learning Smalltalk more quickly and with much less anguish than the other students. It’s certainly how it should be because presumably the other [students] are being
dropped rather cold into a shocking new paradigm and my guys have been dipping their toes into that paradigm for three months.

Ross introduces his students to the new paradigm by presenting them with tempting little tasks ... where it would feel artificial to do a lot of design first. Before our second interview he showed me two LISP textbooks which epitomize his sense of two opposite ways of teaching programming. The one he likes introduces concepts by giving a hint. The other teaches syntax in a step-by-step manner. The projects and lab assignment component of Ross’s course makes up 55% of a student’s final grade and, as the outline states, there is “extensive hands-on practice”. Ross gives his students paradigm-jarring assignments that will challenge not just how they code, but how they think about computers. He also offers some assignment questions that could be done in the style of either paradigm to see whether they’re changing methodology or not.

Ross mentioned one of the standing jokes that he and his students share.

One of them comes to me and says, “Hey, [Ross], in LISP can you do …”, and I say, “Yes”, you know, before I hear any more of the question and of course after a while that becomes a cliche and so they start to ask it as “How can I do X?” , but I think that’s an important switch in mindset. Because I think in COBOL, you come along and say “[Ross], can I do X?”, the answer might be no ...

... But the thing about that question with LISP is when they come along and say, “[Ross], can I do X?”, any X, including X’s that I’ve never thought of before and they’ve never thought of before, they’re dreaming it up out of the air, dancing carrots, the answer’s still yes.

Ross has fairly intimate interaction with his students. They are pretty close and spend a lot of time together. He talked about [taking] them to rather interesting places. Ross seems to act as a guide to his students and has the sense as they work together of swimming through this sea of alternatives influencing them by pulling

What Teachers Know about Programming
Ross cautioned that as an instructor he has agendas other than teaching programming for programming's sake. *Part of what [his] students learn when they think they're learning programming is learning how to ... market themselves and their product. How to give little tempting nibbles and then withhold ....* In a sense, he said, what he does is *anything but teaching.* He uses his students' projects to still indulge in creativity for ... a design of [his] own although it's not [him] at the keyboard, which it used to be. A lot of [his] programming now is done with their fingers. He questioned why he is teaching LISP when almost none of them will use LISP on the job.

Debbe: So by what stretch of the imagination is it worth it?

Ross: ... I can use LISP to encourage them to think about programming in a fairly drastically different way from what they're being told in other contexts and I am sure as I can be that the ones who catch on to it are a much different category of programmers. I think it will stand them in good stead as languages evolve more and more towards a LISP-like model as languages are doing.

Ross says that part of the high he gets comes from watching students catch on to the LISP-world phenomenon. Not all of the students do catch on. Ross described two students in particular, one of whom got high marks in his course, but was *totally without creativity.* She would apply the material *mechanically, rather than using it as something to fly with.* She would always *stop at one level.* The other student is *probably the best* programmer Ross has seen. He learned all the syntax and then *built something of his own and then with that built something of his own and with that built something of his own.* Ross said that the latter student is excellent in the context of the kind of programming that involves *rapid prototyping of things that are new.* However, a colleague who taught the same student PL/I, described him as *a nice kid, but ... not much of a programmer.*
Ross feels that students who get the AI bug

*come to [him] out of the box with a certain mindset, a certain way of looking at the world, such that they don't need a lot of processing on [his] part. They're ready to hear any messages that [he's] got ready to give.*

According to Ross, getting the AI bug is a process. When he mentioned a current student whose head is getting the right orientation, I asked him what he was measuring to see if the students were making the transition. He said that he looks for fluency in lingo and concepts and at their choices of coding applications. As well he listens to the questions they ask and looks at the books they are reading in order to get sort of a sense of general orientation.

If he were to select students for his course, he would interview them looking for those who can make leaps ... to various meta-levels. He would look for signs of life, signs that there's a mind in there. The kind of person he is looking for lives in a world where one reads and one is attuned to intellectual issues in some sort of abstract pattern matching kind of way and is already excited about [artificial intelligence].

Ross mentioned several times that he is unsure whether the talent required for programming is hardwired or programmed. Whether it's some kind of innate capacity they've got or whether they have just been preprogrammed by somebody or something else, who knows? Neither does he understand why some catch on and some don't when all are getting, [he thinks], similar feedback from [him].
An Integration of Ross’s Knowledge

It's a big world. There's room for everybody. To Ross, the world of programming is made up of smaller paradigm worlds which sometimes overlap and sometimes are strikingly different. He feels that there is a major difference between LISP-world and other worlds in terms of working and learning, and in terms of creativity and control. LISP-world, to him, is a place where he can experiment, where he can have an idea and try it out without a lot of pre-planning. It is a place where he can use his control over the inner workings of a language to create things out of the air. In contrast, the world of block structured programming is a place where creativity is constrained by specifications and language structure, where one plans extensively before implementing ideas.

Ross feels that different people work well in each paradigm. His sense is that certain languages fit certain people. Students, he said, come to him with certain mindsets, innate or learned, that predispose them to different paradigms. If he could, Ross would use his feel for LISP-world people to screen incoming students, looking for those that can make leaps ... to various meta-levels, that look at the world in an abstract pattern matching way. Ross sees students of both types in his classes and notices that those that he thinks are good are not necessarily thought of in the same way by his colleagues who teach other programming styles.

Ross's sense is that the switch from one world to another is a difficult one. He had a hard time making the transition and he sees the same difficulty when students are forced to learn new paradigms. He sees that students struggle when they are dropped into a shocking new paradigm. He attempts to help them by presenting them with tempting little tasks and by giving them paradigm-jarring assignments to

What Teachers Know about Programming 75
change not only the way they code, but the way they think about computing. Ross gives students the opportunity to learn in the way that he likes to learn, by doing.

Ross likes to manipulate the computer at a low level, but not in a detailed constrained way. In the same sense, he manipulates his students' world, not by prescribing rules, but by offering more subtle enticements; by pulling strings and pushing buttons, knowing that some will get the bug. He attempts to immerse them in a world where one experiments, where no step by step rules are specified, but where a guide is available to keep an eye on one's orientation and to give hints about new ways in which one can learn to fly into unknown territory.

The creativity that Ross once expressed by writing code and then by designing it, he now finds by working with his students. Although it's not [him] at the keyboard anymore, he can still create dancing carrots on the screen.
Lorne

*a chance to play*

Lorne's educational and employment background is extremely diverse. He has studied Educational Psychology, holds a BSc in mathematics and plans to do an MBA in Decision Support Systems. He has credit for many undergraduate courses in addition to those required for his BSc because, as he said, he has *just taken stuff from all over the place to kind of sample it*. Lorne has no formal computer education.

Before entering the computer field, Lorne held a wide range of jobs: from sewing machine mechanic to cardiac care orderly. Lorne has worked in the computer industry for 14 years, doing customer support, programming and consulting. He programmed in Business BASIC for eight years, but has also worked professionally with FORTRAN, COBOL, Assembly language, Smalltalk and C.

Lorne's teaching background is also varied. As an undergraduate, he did teaching practica in elementary schools, taught calculus and linear algebra labs and worked as a computer lab proctor. Since then he has taught a variety of computer topics: classes in word processing, spreadsheets, database and operating systems; courses in database and decision support theory; and Smalltalk, C and assembly language courses. Currently, Lorne is working as a teaching assistant in a Smalltalk course and a C course.

Lorne is familiar with many computer languages. In addition to FORTRAN, COBOL, C, BASIC and Assembly language (procedural languages), and Smalltalk (an object-oriented language), he has also used PROLOG (a logic language), and LISP (a functional language).
During our interviews, Lorne told many detailed stories about his past and often cited industry experiences to support his arguments. He frequently used examples to illustrate the points he was making and offered such detailed explanations of the object-oriented paradigm (including detailed diagrams to illustrate his coding examples) that at times it felt as though he were teaching me Smalltalk. Lorne used several physical metaphors in his speech, talking about taking a step in C, breaking problems down and playing with programming languages.

**Lorne's Knowledge about Programming**

Lorne likes using the computer as a tool, as opposed to studying computing for the sake of advancing knowledge about the computer. He is mainly interested in decision support systems applications. In keeping with this utilitarian view, Lorne categorized the languages he knows in terms of the applications for which they are used: business, scientific, technical and artificial intelligence programming. The languages Lorne most likes are in the latter section -- languages like PROLOG and Smalltalk.

Because Lorne sees the computer as a tool, he likes languages in which he can quickly and easily develop correct code. Thus he sees great power and flexibility in a language like Smalltalk, which has a large library of easily modified reusable code. Lorne said that as a professional programmer he spent a good 75% of [his] time redoing something that somebody else had already done. He stressed that an advantage of Smalltalk is that the programmer can use and modify existing code rather than having to develop it from scratch. This leads to fast development time and less room for programmer error. Lorne told a story about an intelligent database project he worked on:
I did the prototype myself in Smalltalk in six weeks ... the prototype was 90% of the job and then [the code had to be converted to C] so myself and two part-time programmers spent the next year taking what I had done in Smalltalk and converting it into C. And we didn’t get done in time. We didn’t even get everything I did in six weeks by myself.

Lorne feels that part of the reason the conversion took so long was that he and the other programmers had to create in C the interfaces that are built into Smalltalk. As well, he encountered problems with integration of the C code that did not arise in the Smalltalk environment.

Lorne sees strong distinctions between the programming paradigms with which he is familiar. He said,

> When programming in the procedural paradigm the programmer learns to organize everything, to know exactly what you’re going to do, to have the structured detailed outline, a flowchart of exactly what you’re going to build and know what you’re going to put together, then write all your modules, put in stubs and everything, link it all together and you’re done.

Procedural code doesn’t really look anything like what’s going on [in the world] ... it’s not an exact model.

Object-oriented programming involves a different approach. Object-oriented programmers think about *real world objects*, their attributes and behaviours. They design interfaces by drawing pictures and *keep adding pieces as [they] go along.*

With an object-oriented approach you’re looking more at an exact model.

Lorne thinks that the shift from a procedural to an object-oriented paradigm requires a radical change of thinking. When learning Smalltalk, he *had to unlearn everything [he] knew about structured thinking ... instead of doing a top-down analysis, [he] had to start thinking of things as objects.*
Lorne likes Smalltalk and is also strongly attracted to the logic of the declarative language PROLOG, perhaps, he thinks, because of his strong math background. He said, *Somebody wrote that language the way I think.* When coding in PROLOG, though, he still found that as soon as [he] turned [his] back, shall we say, and stopped concentrating really heavy, [he'd] find [himself] doing something that [he] had done in traditional programming and again [he'd] get into the procedural format ... [he] had to ... make [his] thinking change.

Lorne writes object-oriented code in a *mix* of procedural and declarative styles. He said, *it's declarative in that I have an object and I'm sending it a message, ... but I'm still going to have to ... use some kind of a procedural method.*

In general, Lorne finds that when he learns a new language he at first writes code in the style of a language he already knows. As he learns the new language, though, he starts *adjusting* [his] thinking. He told a story about a Smalltalk assignment that his students were doing. At first, he said, the approach he took was procedural. *I was thinking like that and then I stopped and I said, “But, wait a minute now, we’re doing it in Smalltalk”, click, and my brains stopped thinking.* He then started thinking about the problem in an object-oriented way. I wondered aloud about the influence of the language:

> Debbe: ... *So in that sense the language shaped*
> Lorne: *My thinking.*
> Debbe: *the way you were thinking.*
> Lorne: *And, and yet, that’s the same way I would do it in C++, exactly the same way. Whether it was Smalltalk or C++.*
> Debbe: *Mm-hmm.*
> Lorne: *So I’m not so sure if we’re thinking languages or paradigms at this point.*

*What Teachers Know about Programming*
Despite his sense that going from a procedural to an object-oriented paradigm is like getting out of a car and climbing into a jet, Lorne feels that a good understanding of assembly language programming is important for all programmers. As he said,

... if you understand how the hardware of the machine works and how to program it at that level then [other languages are] just expansion on [assembly language]. [Assembly language], this is the fundamentals. Everyone should know this. Cuz we should know how the machine works.

Languages, he said, have the same general type of underlying structure, so I know how to grab a steering wheel on a car and I know how to grab one in a jet and I'll experiment with it a bit.

Lorne feels that the computer industry is shifting towards the object-oriented paradigm. Its power and flexibility is going to put us very quickly ... in an environment where programming is done more as a systems analysis. Eventually we should be able to stop programming and start designing systems. That should be the end result of this object-oriented approach.

To Lorne, programming is creating something. We talked about the creativity involved.

Lorne: You cannot without imagination, just sit down and write good code. ... the really creative programmers ... [are] like people writing symphony orchestras ... or pieces ... the violin is coming in ... and now we have the oboes ... and they get excited about it ... you can just see the whole orchestra playing ... it's an art ... like mathematics is an art.

Debbe: Yes. So programming to you is

Lorne: It's an art form.
Lorne also described programming as a *thinking person's game* that takes a *lot of intelligence*. In addition to the analogy of the symphony, Lorne compared programming to writing a research paper. He described a person writing a paper:

> She starts writing notes and then she looks at all the notes she's put together with the outline and then she starts fleshing it out, rearranging things and she prints it off, reads it, goes through and changes everything, goes back, keeps modifying it, and keeps iterating until she's done. ... it sounds awfully familiar, doesn't it?

Good programmers, Lorne feels, must most importantly be motivated to want to program. The person he described in the above quotation, although she would *make a darn good programmer* in terms of skill, *would make a lousy programmer* because she is disinterested. Programmers can organize their thoughts into an *overview of an entire system*, ... break it down into all the component pieces, solve the pieces and put the pieces back together until [they] *have a solution to the entire problem*. Programmers think logically. Lorne mentioned an artist he knows who would not make a good programmer because she is *not really aware of cause and effect*. Although he does feel that programming requires creativity, the kind of creativity it takes to draw or paint is *alien to what [he] does*.

Lorne pictures programmers as slightly eccentric people, who like to do esoteric programming, but who know that they *have to do the grunt work in order to get a chance to play with things*. He distinguishes two kinds of programmers, the brilliant ones, who have *innate skill*, who *just sit down at the keyboard and all of a sudden, boom, it becomes part of them*, and those who *plod along* relying on patience and determination. He said that he is of the latter type, although he added that he is *what they used to call a hacker*.

Lorne entered the programming field *by accident*. He had no programming knowledge and was working as a data entry clerk when his company's programmer
quit and he was asked to take over. He struggled on his own for several weeks before the company hired someone else. Lorne continued programming, still learning on his own, reading program listings and manuals, this time with someone to guide him:

I'd start writing stuff and I'd get stuck and I'd say, ... "what do I do?" And [the boss would say], "Read the manual." And then he'd say, ... "if you look at this ... you might find something interesting". And I'd go look in the manuals, and it would guide me right into the area that I should be moving in. So in the next year and a half I learned by writing their system.

The lessons learned in his first job epitomize Lorne's approach to programming. He learned that he was smart enough to solve programming problems and that supporting information was available if he knew where to look. Lorne feels that the most important thing his boss taught him was, I can program anything, anywhere. Any computer language you want to name, I can learn.

Lorne is very confident about his ability to learn programming languages. He said that he has played with virtually every computer language that's around. He seems to learn languages best when he can, in fact, play with the code. He learned Smalltalk by going through the tutorial examples in the manual, [ripping] them apart step by step until [he] knew what was happening. He likes the fact that Business BASIC in interpreted mode allowed you to go through, examine variables and step through a program and actually see what was happening while the machine was changing states. As well, supporting documentation is important. Lorne mentioned several times the amount of time spent looking in manuals when coding in various languages and stressed the importance of knowing the routines available in code libraries.

What Teachers Know about Programming
Lorne's Knowledge about Teaching Programming

Lorne enjoys teaching. After having some difficult experiences with elementary school administrations, he realized that he wanted to teach at the post-secondary level, where he would have the freedom to be as weird as he wanted to be, still be able to teach people and have a lot more time available for things like research.

Lorne said that quite a few of his computer students have a natural talent for programming. Those people you don't even really teach, you just show them where the ideas are and say OK, go at it. In order to learn to program, he feels that students need to sit down in front of a computer and play with it and try things and experiment. He sees himself as a resource person for the students, someone with whom they can discuss solutions. He also stressed the importance of other resources. Several times he mentioned that students really have to know the libraries in order to code in Smalltalk.

Lorne explained how in a Smalltalk course he would implement his thoughts about learning. He suggested that rather than teaching them some basic fundamentals and then getting them to do projects like we would do in another language, which is how they way in which the course is now being taught, he would have students go through the Smalltalk manual, doing the tutorials and having someone explain the examples to them. He said that now the students don't take time out to sit down and play with all these different things and find out. They don't experiment with it. Because that's how you gotta learn this system, is experimentation.

When explaining Smalltalk code, Lorne drew pictures like the one in Figure 3, which incorporates circles and arrows to illustrate objects and classes, as well as detailed code segments which are contained in methods. As Lorne explained the

What Teachers Know about Programming 84
code, he gave very detailed information about the syntax and meaning of each symbol.

```
collections

Dictionary

sort *(self values) asSortedCollection:
[ :a :b | a getDept < b getDept]
```

Figure 3. Lorne's Conceptual Model

Lorne feels that students' first language should be assembly language.

**Lorne:** ... it comes down to a case of understanding the machine and how the machine works. And if ... a person really, really understands the inside of that machine and what's going on inside of it and then starts looking at a higher level language and thinking of all the statements in that language as just ... groups of machine-level code, ... then one starts to understand how a programming language works. And then you start looking at ... a new language ... you will recognize that it has to do certain things in a certain fashion regardless. ...

**Debbe:** So you would say that ... if they have that understanding of that architecture then they will find other languages

**Lorne:** I think easier.

Lorne thinks that he might be able to predict whether a person will be a good programmer, although he has never tried.

**Debbe:** If you think that you could perhaps, what kinds of things would you be looking at?
Lorne: Well, I'd be looking at ability to define the problem.

Debbe: Mm-hmm

Lorne: The ability to figure out some kind of an approach, some kind of means of solving that problem. Whether it was right or wrong, as long as they tried to solve it. If they tried to break it down in any way whatsoever, isolate a part, portion of the problem and resolve that, then I would think they would probably have a darn good chance.

Lorne was surprised to find that a student who he felt, because of poor English language skills, would have great difficulty in a C programming course, is doing very well. He feels that the student is succeeding because of hard work; because he has spent time playing and experimenting. Lorne feels that students can, with patience and determination, succeed at programming whether or not they have a natural innate skill.

Lorne’s students, he said, are having a lot of difficulty learning Smalltalk.

We’ve taught them this top-down structured analysis approach for a couple of years and we’ve actually, I think, in some ways required them not to think, but rather to follow a program that we have instilled in them and then they come into something like Smalltalk, where traditional thinking will mess them up really badly. The more structured they are, the bigger problems they’re going to have cuz they have to stop thinking of programs as modules of code linked together that manipulates data whereas they have to start thinking of real world objects.

Students, he said, have particular difficulty with the concepts of inheritance and polymorphism and with the idea that objects exist in the Smalltalk space. Learning the language is causing them a lot of difficulty because it is a brand new paradigm, a total different way of thinking.

Lorne and I discussed the possibility of using C++ rather than Smalltalk to teach object-oriented programming. Smalltalk, he said, is the purist’s object-oriented programming language because it forces the programmer to use the paradigm. C++,
an object-oriented extension of C, allows the programmer to use the procedural or the object-oriented paradigm. Lorne suggested that although Smalltalk is nice because it forces them to program in that object-oriented paradigm, learning C++ might be a better way to introduce the students to a new paradigm. Because C is familiar to them, the switch would be gradual and he, as a teacher, would be able to be subtle about the changeover. He said that some of the professional programmers he has seen working in C++ were doing a lot of sloppy things, cuz they were doing traditional C programming. He felt, though, that programmers would become good object-oriented programmers in the same amount of time using either C++ or Smalltalk, but that those using C++ would experience less frustration.

**An Integration of Lorne’s Knowledge**

Lorne uses the computer as a tool. As such, he organizes language categories in terms of application and is attracted to languages that make it easy for him to write applications quickly and correctly. His sense is that the computer industry is moving in the direction of the object-oriented paradigm, which offers programmers a large library of existing code that is easily modified. He therefore feels that it is important to teach students the object-oriented paradigm because if [they] don’t know how to program in OOPS (object-oriented programming systems), they’re going to be having a hard time finding work.

Lorne likes to use the computer as a tool to model problems in Decision Support Systems. The idea that it is important for him to be able to model the real world is evident in his choice of examples and illustrative anecdotes. Lorne’s background in computing is totally experience-based. Thus his sense of computing comes from industry examples and requirements. His fascination with the object-oriented
paradigm stems partially from his perception that it allows the programmer to more accurately model the world than does the procedural paradigm.

As well as a need to model the world, Lorne demonstrates a strong attraction to details. During our interviews he described in extensive detail his industry and teaching experiences. When explaining the differences between paradigms, he invariably offered specific examples and explained them in detail. He seems to understand and learn topics best when he can, as he did with Smalltalk, take examples and [rip] *them apart step by step*. His feeling that students should learn assembly language as a first language appears to be a manifestation of his feeling that knowledge of details is crucial to understanding.

The conceptual model that Lorne uses incorporates both his sense that a computer language can be used to model the world and his preoccupation with details. When explaining code, he at first drew circle and arrow diagrams to illustrate his models of objects and classes. Then he added to them perfectly detailed code. The model seems to parallel Lorne's sense that statements in a high-level language are *just ... groups of machine-level code*, regardless of the paradigm of the high-level language. As well, it graphically illustrates Lorne's sense that when doing object-oriented programming, he is using a mixture of procedural and declarative styles.

Lorne sees major differences between coding in the declarative style and in the procedural style. In the former, he is telling the computer *what is to be done* and in the latter he is telling it *how to do it step by step*. The object-oriented style for him is a mixture of the declarative and procedural styles, and is primarily concerned with modeling objects, their attributes and behaviours. Whereas the procedural style is epitomized by structure and planning in a top-down way, the object-oriented style is bottom-up, involving the addition of *pieces as you go along*. It also allows the
programmer to plug in black boxes of already existing code. Lorne feels that the shift from one paradigm to another is very difficult and his thoughts about teaching show his desire to make the transition as easy as possible. His sense that C++ would be a better vehicle than Smalltalk reflects his feeling that the switch should be made in as subtle a way as possible so that the students experience little frustration.

Lorne finds that when switching from one paradigm to another his first tendency is to code in the style of the paradigm which he knows. Thus he feels that the paradigm has shaped the way he thinks when programming. He sees the same symptom in his students when they are learning a new paradigm. However, he also recognizes that certain languages fit certain people. PROLOG, as he said, is written in the way [he] likes to think.

To Lorne, programmers have a certain creativity and are often eccentric. He sees both brilliant programmers, those with an innate talent for programming, and plodders, those who get by with hard work and determination, in his classes. Although both procedural programmers and object-oriented programmers need the ability to break a problem down into components, solve the components and put the solution back together, Lorne contends that highly structured thinking and traditional procedural approaches will be a hindrance to object-oriented programmers.

For Lorne, programming is learned by experimentation. He has no formal computer education, but was immersed in a programming environment where he was forced to learn on his own. He suggested that students learn best in the same way. He also stressed the importance of support. Programmers, he feels, spend a lot of time using manuals and resource people. As well, they should have available to them extensive libraries of code. Thus, Lorne would prefer to teach Smalltalk by
explaining to students in detail the tutorial examples in the manual and by giving them a chance to play.
Isaac

tying all the pieces together

Isaac and I have sat at neighbouring desks since he began his current job approximately four years ago. We have taught the same courses and are familiar with the same students. That we previously had discussed teaching methods and approaches was evident in the tone of our conversations. Because of our shared experience, as we examined the teaching of programming we were able to concentrate on underlying conceptions rather than on superficial details. Isaac wanted very much to discuss teaching, and said that he liked the second interview (in which we concentrated on teaching) much better than he like the first (in which we concentrated on his background). Near the end of the first interview, the following exchange took place:

Debbe: All right, let's talk a bit about
Isaac: We haven't talked about teaching yet.
Debbe: I know. Shall we talk about that?

Several times during the interviews Isaac prompted me in this manner, seeming to signal me that he had nothing more to say about a particular topic or that he wanted to move on to a new topic.

At times Isaac cautioned me that we were dealing with issues about which he had only surface knowledge and commented on his lack of preparation for some of the questions. After the first interview, Isaac sent me a lengthy electronic mail message outlining the details of his past experiences (about which he said his "recollections [during the interview] were flawed") and stating his views about teaching programming. The contents of that message have been included in this section.
Isaac learned to program in the trenches, through night school and correspondence courses, and on his own. His first exposure to programming was in an electronics course. Then he bought and learned to program a personal computer and a programmable calculator and began taking language courses: in FORTRAN, Pascal and assembly language. His work experience includes assembly language maintenance of FORTRAN programs and industrial automation programming. As well, he taught himself C, Modula-2, and some Ada and FORTH. As Isaac said, he has taken courses in everything under the sun and is about to begin a post-graduate degree in computer science. Isaac has written several articles for computer trade journals.

Before starting his current job, Isaac delivered industry training courses and taught electronics (computer architecture and some assembly language programming) for five years at a community college. Isaac initiated the two C language courses at his present school and the bulk of his teaching has been in that area. He also has taught Pascal and computer architecture and established a data communications specialty. Currently Isaac is the head of the microcomputer area and teaches C, OS/2 and network programming.

The languages with which Isaac has worked have been procedural. However, he has had some exposure to LISP (a functional language) and PROLOG (a logic language) and is currently learning C++ and Smalltalk (both object-oriented languages). He plans to spend more time learning Smalltalk in the coming months.

Isaac incorporated a large number of quantitative details in his conversations with me. He told me that 60 to 70% of his teaching has been in C, that he could probably predict with 85% accuracy whether a student would make a good programmer, and that to be a good programmer a person probably should have a minimum
intelligence quotient of somewhere around 110. Isaac rarely used acronyms in his speech. He expanded words like Presentation Manager and Applications Programming Interface rather than calling them 'PM' and 'API'. During our discussions, Isaac used physical metaphors -- saying things like tripping me up, on my feet, and get my hands on -- as well as many cliches, such as neither fish nor fowl, part and parcel and from ground zero.

Isaac’s Knowledge about Programming

To Isaac, programming is a large highly technical subject area which requires very specialized skills. As he said, ... there’s a fairly large body of knowledge that you have to assimilate ... you have to be able to remember all the keywords in the language. If it’s a library that you’re working with, you have to be able to remember what all the function calls are. Programming projects in the real world involve many people and each program is so huge ... you couldn’t possibly remember all of it at once. Isaac is currently working with the Applications Programming Interface in OS/2, a complex system, containing many data types, variables, messages, and functions, in which a lot of the parts of the systems interact. Isaac said that despite nearly two years of steady work, [he is] only now beginning to feel that [he has] command of most parts of OS/2. Programming, for Isaac, means tying all the pieces together. Long debugging sessions, which Isaac said are part of even the most well planned and well designed project, often mean, for him, spending hours and hours and hours poring through thousands of pages of documentation to try and find one little fact. Isaac appears to need to understand each little fact in order to understand entire systems. He liked dealing with bits and bytes in his industry jobs and, when explaining a C program to me, made no assumptions about code behaviour without checking every detail.
Isaac sees programming as an interesting intellectual pursuit that gives him a feeling of accomplishment. He likened it to carpentry.

Debbe: Do you have the same feeling about carpentry?
Isaac: Yes. Yeah, you bash a bunch of things together.
Debbe: Mmm.
Isaac: If the nails don't fall out, you feel pretty proud of it afterwards. So, yeah, it's a feeling of accomplishment.

Programming also involves creativity. The mental activity involved is creation of some sort ... it's like painting. The creativity must be tempered, however.

Debbe: So programming to you means
Isaac: It's a creative pursuit. To work well, it also has to be a disciplined pursuit ...

Isaac often stressed the need for discipline. He said programmers need to be disciplined in order to plan programs and go through the ... long debugging sessions.

Isaac likes languages that impose discipline on the programmer. Thus Modula-2 is his favourite language because its modularity is enforced, because its explicit block syntax means there's less likelihood of ... using shortcuts that end up with sloppy coding techniques, and because it is type safe (that is, an attempt to convert a variable of one data type to a disparate type will cause a compiler error). Isaac said that although C does not enforce discipline in the same way that Modula-2 does, the culture of Modula-2 has infected [his] C programming. He thinks that good C code should look like good Modula-2 code.

Isaac categorizes languages into somewhat overlapping paradigms. Smalltalk and C++ are object-oriented, but have a significant procedural characteristic to them. Modula-2 and Ada, although classified as procedural, have some object-orientation available to them. PROLOG and LISP are in a category by themselves, as is FORTH,
which Isaac feels is kind of a unique paradigm. Procedural languages are, for Isaac, much easier to categorize. He further categorized the procedural languages he knows in two ways. Pascal, C, Modula-2 and Ada are structured, whereas COBOL, FORTRAN and assembly language are unstructured. (Isaac said that he knows at least four distinct assembly languages and itemized them.) Within the structured class, he further specified that Modula-2 and Ada are suitable for big projects because much of the discipline is imposed by the language rather than having to be imposed by the programmer whereas Pascal and C are not.

Isaac took one course in artificial intelligence in which he wrote a PROLOG interpreter in LISP. About the two languages he said,

I didn't like either one of them very much. I actually wrote a little PROLOG interpreter which was not very good and not very successful and I didn't really understand what I was doing ... so of all the languages that I've had contact with, those are the two that I feel least comfortable with.

The discomfort was caused, he said, partly because the paradigm was so completely different. ... partly it wasn't the language. Partly it was I was trying to do something with the language that was unfamiliar to me. Isaac pointed out that the fact that he does not want to pursue use of PROLOG and LISP is not a weakness of the language, rather it is a weakness in [his] understanding and appreciation of the language.

Isaac does appreciate the object-oriented paradigm and said, we're going to require probably fewer and fewer programmers ... now that development systems are getting more intelligent -- object-oriented systems, ... where you can put together an application in ... weeks instead of years. He sees great potential for the object-oriented paradigm and particularly for a language like C++ which supports both procedural and object-oriented programming. Although when coding in C++ he [finds himself] constantly falling back into the ... structured, procedural paradigm ...
[he sees] potential for the mix of the two paradigms. He thinks that both paradigms are necessary because one paradigm is not sufficient to solve all problems. Certain problems, before you even try to take them apart and figure out how they're gonna be solved, the problems themselves fall more into a procedural type of solution. Others, like those he deals with when using OS/2 and Windows, would work very well with an object-oriented paradigm. He said that the object-oriented support which allows him to build a new capability by taking an existing class and adding to it, so you don't have to kind of ... bailing wire methodology to build it, ... [is] a big advantage.

We discussed the influence of language on the way in which a programmer thinks.

Debbe: Some people claim that the language that we code in shapes the way we think. Other people say that we choose languages that we like based on the way we think. Do you have any comment about either of those?

Isaac: I think both are true. I think that oftentimes people don't choose their language -- that their language is mandated by their job.

Debbe: Yes.

Isaac: And once it's mandated by their job, they start to think that way. I think that people who do have the freedom to choose their language probably do choose the language that best matches the way they think. I think a lot of it has to do with, what language were they first exposed to? And if you were first exposed to, let's say, a functional paradigm, you'd probably have a harder time with an object-oriented paradigm or vice versa.

Debbe: ... Do you think that that has any bearing on the discomfort we both felt when going to PROLOG?

Isaac: Yes, oh, for sure, for sure. We and hundreds of others like us have trained our minds to think of problem solutions based in a certain way of doing things, so suddenly having to solve a problem in a way that's foreign to us is going to be very difficult. Which
obviously lends strength to 'we think in the way that our languages deal with problems'.

Isaac feels that the only way one can choose a language that fits one's way of thinking is to become good at many different languages. He thinks that it is possible to become proficient in many different paradigms, but not all at the same time and said that it takes about two years to become proficient in a language.

To Isaac, programmers exist along a continuum from the scruffies to the neats. The scruffies are the hard core, quintessential programmers -- they have long hair, are unkempt, wear blue jeans and sneakers, and spend most of their time writing program code. The neats, on the other hand, wear three-piece suits and project a corporate image. The latter are maybe the [guys who are] doing some of the ... higher level thought work, but, as Isaac said, the dress is ... kind of irrelevant in a way ... it's more indicative of ... an image that you have to project ... or that you choose to project. More significant, Isaac feels, is the measure of quality that he uses to distinguish among programmers. Those with whom he is impressed have broad technical knowledge, apply a scientific method to their thoughts, and produce many lines of good, well documented, zero-defect code a month. Others have put themselves in kind of a narrow box where they [are not] learning new things ... [they are] plodders who bring down that average ... line of code per month figure.

Isaac's Knowledge about Teaching Programming

Teaching, to Isaac, is performing. He feels that in order to teach well, one must be well prepared and interested in the material. That is, one must feel comfortable with the material, have the facts straight, and also feel enthusiasm and excitement. A good lecture, he said, is when [he gets] a flow going, when ... the words that [he says] seem to lead from one point to another. Isaac said that he often cannot predict when a lecture will go well:
As much as I plan my ... lectures -- and I have fairly elaborate overheads that I follow -- ... I still talk more or less off the cuff and just use the ... lecture notes as ... guideposts and when things flow from one topic into the other, well, then the lecture feels to me to be right. ... I consciously try to cultivate that flow, but it doesn’t always work and sometimes it seems like the more I try to cultivate it, the less it works.

Isaac mentioned several times that it is possible that the students do not share his perceptions about lectures that have *bombed*. That a lecture *feels right from* [his] end of it ... obviously ... doesn’t mean much to the students.

Isaac feels most enthusiastic when he is teaching something for the first time. He said, *I’ve often felt that the first time I ever give a lecture is better than any other time I give it. Because I’ve just spent ten hours preparing for that lecture and I’m really up for it and really keen about it.* He enjoys teaching topics new to him and has initiated the teaching of several new courses at his school. The thing he likes most about teaching is learning. In his note to me he said, “the main reason that I am here is to learn -- if along the way I can show a few other people what I have learned, so much the better.”

Isaac feels that programming is taught to prepare students for jobs. Although he thinks that students in elementary and high school levels should learn to be computer literate, that is, that they should learn applications, he said that they should not learn to program because *we don’t need a whole population where everybody can program*. His teaching is geared toward job training. He teaches methodologies that, he feels, the students will require in *the real world*. As well, he said, he attempts to project an image to the students of what the corporate programmer looks like.
Although he attempts to influence the students to see a certain image, Isaac feels that we’re not teachers, we’re just guides, and [the students] have to learn [programming] on their own. As he said,

I think that programming is mainly something that people teach themselves. I think that as much as we can point out the syntax to them, point out the semantics of the language, try and explain to them what’s important, try and get them to understand existing algorithms so they can then build on them, I think in the final analysis that people teach themselves how to program based on a model that we show them. I think a carpenter can teach somebody how to hammer in a nail, but in the final analysis, learning how to hammer in that nail is the person with the hammer in their hand trying it, and I don’t think it’s any different in programming. ... learning programming isn’t learning a bunch of facts. You can learn history by learning a bunch of facts. And as soon as you can mimic the facts you know history. Right? And as soon as you can mimic the syntax of the C language, you don’t know C.

People learn programming through practice, practice, practice. The practice, however, must be accompanied by applied discipline. Practice without discipline limits how far [the students] can go. Isaac's note to me read, in part,

As to my views on teaching programming: more and more I am trying to emphasize “professional practices” above all else -- i.e., good style, good structure, maintainable code, efficiency, and elegance. I feel the syntax and semantics will be easily absorbed by the students with little effort on our part -- the “style” aspect must be hammered repeatedly lest they develop bad habits that they will carry with them forever.

Isaac talked about forcing students to write C in a modular style and about students having an indoctrination with their biweekly exercises. He includes detailed program implementation specifications with his major project to help [the students] and partly ... constrain them a little bit too. One of his major concerns about students learning programming in elementary and high schools is that the students learn a very undisciplined approach to programming. He said that the students sometimes get the idea that the only thing of any value is the eventual code and
that he is always struck how consistently the students seem not to believe us when we tell them how important it is that [structured] approaches be followed.

The content of Isaac’s C course quite closely matches the sequence of the standard C reference guide, written by the authors of the C language. Isaac changes the sequence of topics if the students need certain facts in order to complete a project. As well, he reviews topics, such as precedence rules, that he thinks are particularly important.

Isaac feels that the first language a student learns should be something small, something very type safe, something that enforces structure to some extent. He would rather teach a first course in a very very constrained environment where the language has the capabilities that are needed to teach those structured concepts and no more. Because most of the real work in programming is still done in procedural languages or in languages that are at least partially procedural, the first language should be the best -- whatever that means -- procedural language you can find, which is probably some offshoot of Pascal.

Isaac feels that programmers need above average intelligence so that they can grasp and remember the facts in the large body of knowledge that makes up the discipline. They should enjoy problem solving. Isaac mentioned someone who he thought would be a good programmer who is an inveterate tinkerer. As well, programmers need the temperament to apply discipline to the process. Discipline, however, is not sufficient; the best programmers are able to blend creativity and discipline. Isaac said that someone who was all creativity would be a better programmer than someone who was all methodology, because creative programmers can have discipline imposed on them. Isaac feels that he can predict which students will be good programmers by listening to their speech patterns. Those that do not pay any
attention to [the] grammar and syntax of their language, who speak in slang and ask ill-formed questions, he feels will not make good programmers.

An Integration of Isaac's Knowledge

Isaac's programming background is strongly procedural and he has worked with many assembly languages. His view of programming and programs as an integration of a large number of details is consistent with this experience. His attention to detail is apparent in his speech, and in his need to get the facts straight, not only in order to teach them and to integrate them into programs, but also in order to respond to interview questions. Isaac's sense of programming is well illustrated by his carpentry analogy: programming is building things from smaller parts and hoping that the nails don't fall out. The way in which Isaac categorizes languages also offers a clue to his view of the programming world. His view of the procedural paradigm and the ways in which other paradigms integrate with it has a high degree of granularity. His descriptions of paradigms with which he has little experience are much less precise.

Isaac teaches programming so that his students can get jobs in the programming field and consciously attempts to impart to them his image of a corporate programmer. Programmers are intelligent people who like to solve problems. They are creative, but have the temperament to apply a scientific method to their thoughts. For Isaac, the best programmers combine a disciplined thought process with a broad range of technical knowledge.

Isaac believes that teachers do not teach programming, rather that students learn it on their own through much practice. However, he feels it is imperative that students apply disciplined methodologies that will allow them to cope with the integration of the many small details they will be expected to deal with on the job.
Thus he very much takes the expert approach to teaching programming, attempting to indoctrinate students into the *stepwise refinement disciplined kind of approach* by managing the way in which they do their lab assignments and projects. Isaac's style is also very structured. His lectures are supported by elaborate prepared overheads and course notes and he pays strict attention to language details.

Because most industry programming is done in the procedural paradigm and because the first course should be taught in a *very very constrained environment* in order to impose the necessary discipline and allow him to teach structured concepts and nothing more, Isaac feels that the students' first language should be a Pascal derivative.

Isaac maintains that working in a specific language or paradigm shapes the way in which a programmer will approach problem solving. His preference for the highly structured procedural paradigm is evident in his selection of Modula-2 as a favourite language. It is possible that Isaac's views will change after more exposure to the object-oriented paradigm. However, at this point, his teaching strongly echoes his desire to impose the discipline that will help his students in *tying all the pieces together*. 
Kyle

the power of modeling the real world

Kyle was first exposed to programming 14 years ago in a computer science undergraduate co-op program. The first language he learned was PL/I and his work experience involved PL/I and APL programming at a large company. After working for some time and living abroad for several years, Kyle, while investigating continuing his education, was introduced to the concept of expert systems which immediately caught [his] interest. He completed a college diploma program with a specialization in expert systems and worked for a short time building expert systems in PROLOG. He then began his current job working as a systems analyst, a position he has held for approximately six years. His work during that time has been varied and has included the use of FOCUS, 4GLs, SPSS, REXX, high-level tools for Windows, and Oracle. Kyle also does consulting work using Smalltalk and PROLOG. He is presently working on a PROLOG based knowledge system involving genetics research.

Over the past several years, Kyle has taught various computer courses in topics such as object-oriented programming, knowledge systems, Pascal, LISP, APL, PROLOG and Smalltalk. He is currently teaching Smalltalk.

Kyle knows many programming languages. He said that it would be easier to list languages that he doesn’t know than to list languages that he does. We discussed some of the languages with which he is familiar: Pascal, PL/I, C, assembly language, APL (procedural languages), Smalltalk (object-oriented), LISP and Scheme (functional) and PROLOG (logic).

Kyle seemed very calm and thoughtful during the interviews. Several times during the course of the interview he appeared to reconsider and clarify his opinions. He
used several analogies and referred often to eastern philosophies, with which he is very familiar. As well, he used physical and visual metaphor in his speech: talking about things which are naturally reflected in the object-oriented paradigm, about products which support a certain view and about writing a system to capture knowledge.

Kyle's Knowledge about Programming

To Kyle, programming is a discipline that gives him an outlet for creative expression; that allows him to construct things and to manipulate the world. The creative process is, to him, similar to the process of composing music. However, Kyle likes the fact that programming is more practical than composing and allows him to solve real problems. His current interest is in socially relevant applications of knowledge systems and AI (artificial intelligence).

Kyle said that programming involves disciplined and logical thinking and forces the programmer to think abstractly in order to make conceptual mappings of a problem domain. He enjoys the intellectual stimulation he gets from programming; he said that programming gives him a gee whiz effect and that he finds the rapidly changing discipline challenging.

As well, Kyle sees an aesthetic beauty in programming. He gets a sense of pleasure from this beauty in the same way that he gets a sense of pleasure from a beautiful mathematical proof. There is a correlation, he said, between the pleasure he gets and the aesthetic qualities of a language. Languages which are consistent to some type of theoretical basis are aesthetically pleasing to him. Thus Smalltalk, which is completely consistent to the object paradigm and PROLOG, consistent to the logic programming paradigm, are his favourite languages. He is intrigued by APL, in part because of its consistency to the ... paradigm of ... manipulating matrices. Other
languages, which are more murky, which just are built to get the job done, he responds to less positively.

Kyle likes programming, but does not like it to outweigh other aspects of his life. He said that what he dislikes is that programming can create too much mental activity [so that he does not feel] completely integrated emotionally, mentally, spiritually. He also dislikes programming work which is trivially easy for him. The only area in which he now finds difficult and challenging and interesting problems is artificial intelligence.

Kyle pointed out that he feels that working as a programmer alters aspects of your personality and mental characteristics. He finds that because it is easy to quickly change the malleable objects in the world of software, the transition from the world of the mind to the world of physical things and people is jarring. During the transition, when he is becoming grounded, he is impatient and frustrated with the slowness of manipulating objects in the physical world or the amount of effort required to make a change. With people he is more short, less sensitive and less compassionate than he would normally be. If someone is not doing something right, he wants to just sort of push that button and make [them] ... do it the right way. While he recognizes that, in the real world, the analytical approach is not the only way, he finds sometimes that attempting to resolve conflict with non-analytical people is like a clashing of ... two universes.

Kyle knows many languages, which he categorized by paradigm. He distinguished four classical paradigms: procedural, logic, functional and object-oriented. As well, he mentioned two others, which he said perhaps could be classified as paradigms: a data modeling paradigm and a rule-based paradigm.
The procedural paradigm is one in which you ... are specifying a series of actions .... A program is made up of a series of statements which are executed in sequence and the actions are carried out in the order in which functions and procedures are called.

The logic paradigm is based on the predicate calculus and involves the specification of relationships or constraints. Kyle said that PROLOG is an example of a language which uses the predicate calculus with extensions, such as backtracking and the fact that clauses are chosen in the order in which they have been entered.

The functional paradigm is based on the mathematical concept of functions. Kyle said that although LISP is not a functional language, because it contains constructs such as the go statement, loops and macros that fall outside of the functional programming paradigm, one could program in LISP as though it were functional by limiting the programming to a subset of pure functional programming.

In the object-oriented paradigm, data and ... procedures are bound together into single entities called objects which are organized into a hierarchy of [classes] ... which have the properties of inheritance. Object-oriented programs involve the sending of messages. Kyle stressed that the object-oriented paradigm reflects a very common way that people think about a lot of the world. If you were to look into ways people think about the world in terms of generalization, specialization or whole/part categories, these are things which are fairly naturally reflected in the object-oriented model.

The data modeling paradigm is almost an active data dictionary approach, which, he said, is the de facto standard paradigm for a lot of ... business development tools these days and rightly so because it’s an extremely effective and quick way to develop simple business applications. When programming in this paradigm, one defines a relational model and attaches validation routines to database fields. Screen
generation and transaction processing are done automatically. This is almost a paradigm in itself, being only vaguely procedural and having sort of a flavour of the object-oriented paradigm, but not quite.

The rule-based paradigm has some analogies to the logic paradigm, but is not necessarily based on the predicate calculus. It involves the definition of discrete independent rules or constraints which are then processed through some type of an inference engine, be it backward (chaining), forward (chaining), or a constraint-based engine.

Although he sees similarities among languages in various paradigms, in the sense that they employ some of the same control structures and data objects, Kyle spoke at length about the advantages of the object-oriented model, which he said gives one the power of modeling the real world. He said,

with an object-oriented language or with various AI (artificial intelligence) knowledge representation tools and languages ... you can get a very high-level abstract symbolic representation of the world which has a relatively close mapping to the classical ways that we ... model the world internally.

Other languages, such as C and Pascal, do not allow such a [close] conceptual mapping. As Kyle said, part of the difference [between coding in the object-oriented paradigm and coding in another paradigm] has to do with thinking at a different level of abstraction .... When you do good object-oriented programming or design, you can think at the level of abstraction that models the problem domain more closely.

Languages like C and Pascal do not scale up well in the way that object-oriented languages do. By this he means that these languages

may appear to solve [a] problem adequately without too much programming effort for a small case or a small problem. But
when you try to scale up the problem or extend the specifications, ... it starts to be very unwieldy [sic] to develop the code and do the design.

Kyle dislikes languages which make it difficult for him to achieve a specific goal. The object-oriented paradigm’s *modeling capabilities are supportive of development*. As well, object-oriented languages come with a *tremendous amount of built in code*. The object-oriented programmer *can focus on just what’s new in [an] application -- what’s original -- and use existing components to support everything else*. The paradigm *provides a way for organizing literally hundreds of thousands of ... software components in a way that you can browse through ... and use them and manage that level of complexity and size*.

Kyle often spoke about object-oriented programming and artificial intelligence programming in the same breath. He said that a lot of the concepts in object-oriented programming are also found in *frame-based* artificial intelligence programming and that both are concerned with looking at the problem of representing knowledge.

Kyle’s favourite languages are PROLOG and Smalltalk for different reasons. He has a lot of experience in Smalltalk and likes it for solving *general purpose programming problems*. *It’s extremely flexible and versatile and it doesn’t get in the way*. *It scales up well* to big problems because of its adherence to the object-oriented style, with its modeling capabilities and its support of large libraries of code. As well, Kyle likes Smalltalk’s development environment and gets a sense of pleasure from the aesthetic beauty of the language because of its consistency to the paradigm.

Kyle likes PROLOG for artificial intelligence programming, which, he said, often involves *very sophisticated algorithms that involve pattern-matching or ideally backtracking*, or in which you’re developing *very rich complex data structures*.
PROLOG ... supports the creation of algorithms and complex data structures and their manipulation really easily and elegantly. Kyle said that he has sometimes solved the same problem in several different languages and that in his experience, the PROLOG solution is profoundly smaller in terms of lines of source code and has an elegance or a clarity to the design that is greater than any other language that [he has] worked with.

Kyle feels very strongly that one's thinking about computer problem solving and design is influenced by the language that you've learned, rather than ... by your inherent mental proclivities; that one's previous conditioning and knowledge affect one's understanding of a new language. He gave an example of his first attempt to code in Smalltalk:

I'll give you a good example. The first problem I ever gave myself to do in an object-oriented language was to draw a graphical tree of a class hierarchy on the screen. And when I first implemented it, I did it using one big array, in which I populated -- it was like a sparse matrix -- ... I populated it with the nodes in this tree. Because that was the type of data structures and algorithms that I understood. That's a horrible way to implement a tree. And not at all in the object-oriented paradigm although I did it, like I forced that into the object-oriented language.

Now that he has learned to model in an abstract data type way, he finds that he applies that approach to coding in languages like C or Pascal. It is not that he is making Pascal look like Smalltalk, but rather that he is applying more sophisticated computer design knowledge to ... 3GL (third generation language) programming.

When I asked him to describe to me his image of a programmer, Kyle replied that he doesn't have one, that, to him, programmers are simply people with an occupation. He said that people are like histograms, with many different facets of attributes. I wondered about the histogram analogy:
Debbe: So people that are programmers, do they have a certain different kind of uh

Kyle: Yeah, well peo...

Debbe: What word am I looking for?

Kyle: Yeah, yeah, certain patterns

Debbe: Yeah

Kyle: of histograms (he laughs)

Debbe: But, do their histograms match? If you could take like all the histograms in [this office]?

Kyle said that although not all programmers would share common personality type histograms, some of their intellectual skills show similar patterns. They have strong analytical and deductive reasoning skills. They are good abstract thinkers and are systematic and methodical in planning and in following sets of instructions. The latter skill, Kyle said, is one which programmers have learned through ... painful experience.

Kyle feels that application areas are becoming increasingly abstract and sophisticated and that programming tools now offer extremely abstract levels of usage. He said that

the future of software engineering is going to more and more involve development by browsing existing libraries of code, reusing software components and gluing together existing software components, rather than building systems from the ground up.

Kyle sees obvious labour saving advantages to the browsing approach and said that the object-oriented paradigm is an attempt to address the way in which programming will be done in the future.
Kyle's Knowledge about Teaching Programming

Kyle feels that education takes place ... through dialogue. When he is a student, he said, he will sit right in front of the teacher and draw everything [he] can out of that teacher and interact with them. He said that the cultural conditioning that causes some students to be inhibited about learning in that way is not healthy and should ideally be broken. Kyle feels that one of the attributes of a very good teacher is that they're capable of motivating the students to want to learn. He, however, is more comfortable with students who are already highly motivated and has no interest in trying to motivate students who don't want to learn. He said, I can't see what the point of their lives is. ... In my opinion we should be ... focused and motivated about everything we choose to do. Otherwise don't bother. Kyle feels that motivation should be central in elementary and high schools, but that by the time people reach the post-secondary level they've made a conscious choice about what they're doing with their lives and they understand the power of their actions and the implications of their actions.

Kyle said that it is important for teachers to remember what students can and cannot change. He said that students may show lack of skill due to not being very intelligent inherently, due to lack of effort or due to lack of good learning experiences. Whereas one can change amount of effort and one's learning environment and learning strategies, Kyle pointed out, one cannot change one's inherent intellectual capacity, so most simply it would be very unskillful to tell a student, 'you're stupid, you should try harder'. According to Kyle, the students in his current class appear to be not very skillful. Because of pressures on their time, though, he realizes that they don't apply themselves to the course like they could potentially, which probably has an impact on [his] perception of their skillfulness.
Kyle thinks that programming could be taught in high school for the sake of forcing adolescents to strengthen their mental skills at logical reasoning, abstract reasoning, disciplined reasoning and going through a design ... in a structured fashion, -- skills that are general and useful in many categories. In his courses, though, he is teaching students so that they can use, in a job, the skills and knowledge they acquire.

I asked Kyle about the image he imparts to his students:

Debbe: When you’re teaching these students, are you consciously trying to impart a particular image about programming to them?

Kyle: Yes, definitely.

Debbe: What image is it?

Kyle: Um, there’s probably more than one. One of them is a reflection of my feeling that ... things like knowledge representation and artificial intelligence -- and the types of modeling of the world that they imply -- are rich intellectually exciting and subtle topics that are stimulating and that represent ... a more sophisticated use of a computer that is possible. ... I’m trying to expand their horizons.

Kyle is also attempting to present an image of object-oriented programming to the students, a view that complies with his notion that in the future software development will involve browsing ... libraries of code ... and gluing together existing ... components. A third image that Kyle is trying to convey is that there is an aesthetic beauty in ... a good modeling object-oriented or classical knowledge representation approach to some problem. He is not sure, he said, how well he conveys this last image, finding that it is more or less inherent in the students -- either they have sort of the type of mind that is stimulated or finds pleasure in the abstract aesthetic beauty, or they don’t.
Kyle teaches his current course in a transitional way, changing the course somewhat in response to the students' needs. In their lab assignments, for example, he has students work through chapters in the book, then assigns small problems. The first problem instructions are very detailed, giving instructions like “Click the Accept button”, but later instructions are more general, because Kyle expects the students by that time to have understood the pattern of what things to do. The students' tutorial exercises are not graded. I asked about their response to the exercises:

Debbe: So when you ask questions ... it's just a question for them to think about, not to answer for you?

Kyle: Yeah. Yeah. I would like to think that the students would actually take the time to consider that question.

Debbe: Do you think they do?

Kyle: ... because ... they would be inherently interested in trying to master or learn the subject.

Debbe: Yes.

Kyle: And some of them do, but some of them don't.

Debbe: Yes.

Kyle: I have very little patience with students who aren't inherently interested in learning and mastering a skill.

Kyle assigns major programs to the students in addition to their lab tutorials. His first assignment contains the following note:

This problem is more suitably handled by a DBMS (database management system) or 4GL (fourth generation language) and report writer, than by Smalltalk. However, for the purpose of education, this problem is being given first because it is 'familiar territory' while illustrating many design and development issues common to most object-oriented development.

Kyle said that the students have so many new concepts to learn that if on top of that, you gave them a problem that was more suitable ... to the object-oriented paradigm,
they would probably have to learn yet something else. In doing the first assignment then, Kyle expects that the students focus more on basics like ... 'how do I use the development environment?', 'what's a valid syntactical statement?', ... 'what methods can I use?' The second assignment involves something that is more in the flavour of the object-oriented paradigm.

Kyle's outline contains useful stuff that the students want to know: the major themes to be covered and how they will be evaluated. As well, he includes course objectives because, although they don't usually think about that, he wants them to get a sense ultimately of what [he is] trying to do with them and how far [they are] going to go, what level of skillfulness they will achieve.

Kyle has very strong feelings that the students' first language should be a language which can do a lot instead of a language which can do a little; that it would be better to start students off with the best instead of the worst. He said,

> When the students' first language is a very limited, crude, ... narrow language like Pascal or C, ... for a long time afterwards their view of what you can do with a programming language, their view of what you can do in computing science, is tainted by that limitation. ... If you started off a computer science student with the most powerful, advanced, representation languages -- things like Smalltalk or various AI (artificial intelligence) tools that ... model the world in a closer conceptual mapping than you get with a low-level language like Pascal, I think they'd have quite a different ... creative view about what you could do with a programming language.

Scheme as a first language, he said, is no better than C or Pascal although because it is flexible one could build an object-oriented system in it. PL/I was a good first language in the context of the world of the late 1970's, but like any language like ... PROLOG or LISP or Pascal or C or what have you, ... none of them have a close ... mapping or modeling directly built into them for representing problems in the problem domain the way we often think about them symbolically in our minds. I
asked if Kyle thought that learning Smalltalk first might be detrimental in the sense that students would not understand what is happening inside the machine, the way they would if they learned assembly language as a first language. He replied, Why does a student need to understand what's going on? If what you want them to do is model problems in the real world, why should you have to understand what a register is? There is a decreasing need, he said, for students with bit twiddling skills, and an increasing need for students who can think and model at the abstract level.

Kyle feels that some people have inclinations or behavioural skills that will allow them to be good programmers and that others do not. Maybe after five years of intensive training ... you could turn them into something good, but [they are] not natural programmers. Natural programmers have strong planning and analysis skills. They think analytically and apply systematic methods to problem solving. Kyle gave an example of someone who would not be a good programmer because she doesn’t exhibit skillful analytical thinking. Nor does she exhibit the type of systematic application of method to problem solving or analysis in other problems of her life that you would ... want to find in a natural programmer. The person he was speaking about is an excellent painter. Kyle says that he himself can’t draw a stick man. ... We’ve all got our talents.

An Integration of Kyle’s Knowledge

Kyle appears to look at the world from a very broad perspective. He sees many facets of people and programming, which he is able to analyse and compare. Kyle has a very thorough understanding of the different classical programming paradigms and further distinguishes programming styles by adding his own paradigms (data modeling and rule-based) to the others. Kyle’s ability to integrate many perspectives is apparent in the aspects of programming that he enjoys. Programming is a
creative outlet that allows him to solve practical socially relevant problems, but which also affords intellectual stimulation (if he can find a challenging enough problem) and a sense of pleasure from aesthetic beauty. The languages he most likes reflect his feelings. Both Smalltalk and PROLOG are pleasing to Kyle because of their consistency. Smalltalk also offers practicality due to its ability to closely conceptually map the world and reusability of code and PROLOG offers elegance and clarity.

The images that Kyle presents to his students echo his broad view of programming. He hopes to share with them an appreciation for the aesthetics of programming. As well, he wishes to give them a sense of the practicality of the object-oriented paradigm in modeling and solving real world problems. That programming offers Kyle intellectual stimulation is clear from his feeling that teaching programming at a young age might [force] adolescents to strengthen their mental skills.

Kyle also recognizes that people have many facets. He has no particular image of a programmer because programming is just one aspect of life. He does, however, see common intellectual skills among programmers: strong analytical and deductive reasoning, abstract thinking and planning and the ability to methodically carry out instructions. The sense he has that people have many varying qualities is also evident in Kyle’s comments about predicting programming success. Although certain people lack the natural intellectual skills he sees in programmers, he said, it is because they have not needed to develop those skills, but have developed skills in other areas. Furthermore, he recognizes that analytical skills, while they have a certain power and advantage, are not the whole story.

Kyle seems to feel that others should share his desire to take a broad perspective. He thinks that students should want to learn and that people should be motivated
about everything we choose to do. Thus he attempts to broaden his students' horizons; to change things so that they see the bigger picture.

Kyle recognizes that there are some things, such as inherent intelligence levels, that cannot be changed. However, other things, like cultural conditioning can and should be altered. There exist, he feels, strong influences that can shape the ways in which we interact with and perceive the world. Conditioning from culture, past learning experiences, previous programming languages and even the act of programming itself strongly shapes behaviours.

Behaviours should be changed, Kyle seems to feel, so that people can attain balance in their lives. Thus he dislikes the lack of integration he feels when programming causes too much mental activity. As well, he feels that it is a psychological defect that he feels frustration and impatience with people and physical things while he is becoming grounded after spending a day programming. In terms of teaching, Kyle feels that students have a very limiting view of programming that could be changed to a different ... creative view if they were taught different languages. He wishes to expand their horizons by showing them that you can use a computer for much more than counting beans and by showing them the practicalities of using a language that, through support of reusable code, represents the way in which he believes software development is moving.

Kyle recognizes that it can be difficult to make transitions from one way of thinking to another. Thus, for example, he finds it jarring to enter the physical and interpersonal world after having spent time in the mental world of programming. He sees that students, accustomed to the world of procedural programming, find the shift to the object-oriented world a difficult one. In order to assist his students in making the shift to Smalltalk, he gives them specific detailed instructions until they
are able to see the pattern of how things work. His first assignment is transitional in that he expects them to work on the basics of the language without having to immediately make the shift to a totally object-oriented way of thinking. His methods for smoothing the transition echo his sense of the way in which people model the world internally. That is, they see patterns and model the world in terms of generalizations and specializations. Not only does Kyle wish to make students' transition to the object-oriented world easier, but he also believes that their first language should be in this paradigm, because it gives them a broad view of what programming is about by introducing them to sophisticated applications and utilizations of computers and by giving them the power of modeling the real world.
Mario

wanting to push something to the limits

Mario has taught computing at the post-secondary and secondary level for approximately 15 years. He became interested in computers when he was in high school and, after a year as a physics undergraduate, eventually graduated with a BSc in computer science and mathematics, working at various programming jobs while going to school. Mario's Master's degree studies involved the implementation of LOGO. Through this work he became interested in the use of LOGO in the classroom and decided that he would rather make a career of teaching than programming. He completed a one year teacher training program and began teaching high school math and computer science. Finding that he did not enjoy teaching at the high school level, he returned to university to do further post-graduate work and to eventually accept a full-time teaching position at his university.

Mario has taught a variety of languages: Assembly languages, BASIC, BCPL, C, PL/I, Pascal, Modula-2 and FORTRAN (all procedural languages), LOGO, Scheme (functional languages), Smalltalk and C++ (object-oriented languages). He has worked with many other languages such as COBOL (procedural), LISP (functional) and PROLOG (logic). He primarily teaches programming languages and operating systems and is currently teaching artificial intelligence and logic programming using Scheme.

Mario and I have met many times over the past five years at meetings of a province-wide post-secondary computer educators' committee. We have discussed methods of teaching programming and of predicting programmer success on several
occasions and continued our investigation of these topics during our interview sessions.

The transcripts from Mario’s interviews are twice as long as those of the other respondents. He spoke quickly and at the end of the second interview I let the tape run as we discussed pedagogical issues he had introduced that are somewhat outside the scope of this study. Mario’s conversation was punctuated by many anecdotes, examples and jokes, by means of which he described his experiences and his knowledge of computing research and current teaching practices. He used several personal physical metaphors when describing programming: mentioning students that just get paralyzed when programming, talking about having a visceral sense of what the computer is doing, and suggesting that he feels a little bit nauseous when reading PL/I code.

**Mario’s Knowledge about Programming**

Mario finds a personal joy in programming. That joy comes from the sense of achievement he feels when he solves a problem so hard that, on first examination, he was unsure how to solve it. He gets a thrill, he said, from working through solutions and often finds that by the time he actually writes code he has thought it through and modeled it and simulated it in [his] head so much that he has a very strong sense of what the program is doing. He talked about having a visceral feel for what the computer and the program are doing. When he first began programming, he said, it was very much a sense of the whole machine. The book that changed his life was DEC’s Small Computer Handbook, which allowed him to completely understand the machine even down to the hardware level. At that time, he said, It was very easy to write a program that would take over the whole machine and he liked the element of risk involved in [pushing] things beyond what you’re supposed
to. Nowadays, programming is less about knowing the machine than about having a very intuitive level understanding of programs and programming tools.

The desire to understand and learn the peculiarities of various tools sometimes makes Mario feel frustrated with the tools' limitations and often leads to the danger of going off on a tangent. There exists a tension, he said, between stopping to build or better a tool (e.g., an editor or a library) and getting on with his task. On the other hand, Mario does not find program bugs particularly frustrating. He expects to have bugs and finds that he can back off, switch off the judgmental part of [his] mind and start gathering some data in order to track down their causes.

Mario said that the more [he] looks at languages, the more similarities [he tends] to see. He feels that the same semantics underlie all languages; that they are the same in the sense that ... we can achieve the same result with any of them. However, Mario does find that some languages allow him to solve problems more easily than others. A good language maps the ... domain that you're trying to deal with; it [fits] the kinds of problems that you can reasonably do with [it] reasonably well.

Sometimes Mario has the sense that he is fighting a language rather than using it as a tool. Languages like FORTRAN, BASIC and Pascal have limitations that force him to put energy into things that don't have anything to do with the problem at hand. He is forced to deal with bugs that have nothing to do with the problem space, but rather with paradigms of the language. He cited PROLOG as an interesting example: PROLOG is a kind of a funny language because if you're doing things that PROLOG is designed for, it works amazingly well. If ... you want to do something that isn't really logic oriented, you end up ... saying, "oh, this is really horrible".
As well as seeing differences in the applicability of languages to particular domains, Mario sees differences in the ways in which languages allow him to express ideas. He said that although the thrill of programming is to some extent language independent, what makes certain languages pleasant and others not, has a lot more to do with the sort of mental fit of the languages. In other words, not ... is it Turing complete -- every language is -- but rather whether or not ... you feel like this language expresses your ideas clearly. To Mario, a well-designed language is one in which the rules make sense. His

notion of how you design programs has to ... do with ...
information hiding, encapsulation, object-oriented programming and so on .... A language which supports that in a clean way is going to be much more comfortable than a language that doesn't. Now adding to that, a language in which I can type in expressions and evaluate them, ... [that] will allow me to build a program up interactively, is a more comfortable language than one that requires me to write a huge program and then try it out and see if it works.

Although some languages fit better than others, Mario feels that programmers shape the languages in which they work in that the way one attacks problems with a particular language differs from person to person. You are really using a dialect of that language that is your personal dialect, or corporate dialect or something like that. Thus Mario works in C as if it were Scheme, compiling very incrementally and writing code that classical C programmers would question because that's the dialect of C [he likes] to program in.

Mario categorized the languages he knows in terms of the ways in which he uses them. The OOP languages, PROLOG, Smalltalk, Scheme, LISP and C++, he would use to write serious [applications]. C, assembly language and Modula-2, he said, are not very elegant, but are practical. They are necessities in that often he will use them when speed or system access is an issue. He often mixes C and Scheme, for
example, when it's just not suitable to use Scheme alone. In general, Mario feels that it makes some degree of sense, where you have a language that says you must think of the problem this way, to write programs in mixtures of languages. If the problem isn't really suitable to that decomposition or that paradigm, then sometimes you're better off to simply put pieces of your program in ... different languages and interface them together. Mario's favourite languages are Scheme and Smalltalk, for different reasons. He likes Scheme because it is small, clean and orthogonal. However it does not have the elaborate library that Smalltalk offers. Smalltalk, although the environment is very powerful, is not as clean as Scheme. Mario also likes C++'s libraries, which allow him to leverage off other people's work, although he does not want to look inside the libraries because he will be too sickened.

Mario would select a language for a particular project based on the requirements of the people to whom he was delivering the project: the platforms the program should run on and the features and efficiency level required. He said that the person who wants to choose their favourite language for everything ... often doesn't get things done, just because they're so busy trying to force their favourite language to do things that it doesn't do.

Mario's image of a programmer is the kind of droid who works in the large project in the large enterprise and very rarely does anything creative. However, he said that the kind of environment they find themselves in tends to be one of ... fairly limited kinds of options. So you're not really seeing them, you're seeing their environment.

Environment aside, Mario sees a tension between two distinct kinds of programmers. Neither end of the spectrum seems quite right if you're actually
talking about ... being able to do something that is useful and practical. One type of programmer is the kind that wants to push things to the limits, that wants to explore the boundaries of the system. These programmers are clever, and want to understand the rules and the causes of program malfunctions. Because of their need to understand and because they are confident that they can work through problems, clever programmers spend a lot of time learning, exploring and finding better ways of doing things. An inability to make strategic plans or to evaluate success or failure, however, means that these programmers often do not produce a final product. Mario feels that their productivity can't be measured in terms of lines of code produced. He said that although to him they are good programmers, large programming shops tend not to want to hire them because once they've solved the problem they want to go on to something else and may become resentful if asked to repeat tasks. They have more fun, though, Mario said, than goal-oriented programmers who will subordinate everything in order to achieve a goal. To these programmers, the less known about the system the better. Rather than spending time finding better and easier ways of doing things, they tend to configure a working environment and then leave it alone. By doing so, they get more work done than the clever programmers. Mario feels that it requires exceptional ability to interact effectively with [the] complicated [programming] environment. Programmers must be able to rise to the challenge of being in this flexible environment and still get the task at hand done. As well, they must be able to communicate. What distinguishes my notion of a good programmer from a classic sort of 60's hacker ... is that I believe programmers should produce something that other people can read and understand.
Mario's Knowledge about Teaching Programming

Mario enjoys teaching and appears to take a strong interest in his students' success, not just in terms of skills acquisition, but in terms of how his course interacts with the rest of what makes them competent. To him, programming is a liberal art, learned not necessarily in preparation for the job market, but in order to acquire general principles about problem solving. As he said,

there's a problem solving component there that really has nothing to do with computer science that learning to program can help you deal with -- to do with setting objectives for yourself, deciding whether or not they have been met, learning to cope with failure .... Those kinds of skills ... are skills that an educated citizen in our society ought to have.

Although he is aware that there is no conclusive data to support his claim, he said that it's possible that learning how to program at a young age would actually inculcate those skills. Mario feels that it is easier to teach young children how to program than it is to teach teenagers, because children haven't yet learned that they're stupid and doomed to failure and they should only do what they're told. That really comes in high school, by which time students have developed attitudes, like math aversion, that hinder their ability to learn programming. Although teaching programming at a young age would allow a teacher to address attitudinal issues, Mario questions the validity of doing so, given that students in elementary schools face serious problems such as alcoholism and hunger.

In keeping with his view that programming is a means of teaching general principles of problem solving, Mario stresses analysis over production of end results. His Scheme course is taught as a science course, not as a computer programming course and includes a lab component during which the students carry out formal exercises which involve analysis and explanation of results. The end result, said Mario, is mostly what’s going on [in the student’s mind] rather than the
Thus Mario concentrates on design and semantics issues rather than on syntax. Students, he said, are syntax driven ... therefore it's really important to pull them off that and say, "That isn't important. I don't care if what you say is syntactically correct. The compiler will, but I don't." ... I want to know whether it communicates ... effectively. He expects students to understand in general what program code is doing even if it contains minor bugs.

A good teaching day, for Mario, is one during which he is able to communicate things; during which he has managed to synthesize a bunch of different things at different levels in students' heads, ... not only the language issues, but also design issues, theoretical issues and analytical issues. If classes [go] wrong, that is, if students don't understand how to apply the concepts that Mario has taught, he feels that he has done something wrong; that he has not managed ... to do anything more than give them a bunch of words they could memorize.

Mario feels that pedagogically [it is] nonsense to teach subjects such as physics, computer science and english as if they were dead things that one learns so that one can admire them. Thus in his Scheme course he presents computer science as an evolving organic topic by presenting various paradigmatic models for discussion, evaluation and improvement. Moreover, he feels that it is important for students to know that he, as a programming teacher, does not write perfect code and praises students for finding his bugs.

When teaching a new language, Mario relates it to languages that the students already know. In his C course, students translate a project they wrote in Smalltalk into C. In a sense, he said, he is teaching the students how to write C as if it were Smalltalk, which is a strange thing to want to do, but what I'm trying to do is give them a really solid basis in things like ... abstract data types and so on and saying that
that comes from the object-oriented programming [they have] already done.

Mario's extensive Scheme course outline highlights the sense he has that teaching programming is about much more than skills acquisition. The outline includes a statement that harassment is a threat to the principles of "academic freedom, respect for each other, and equality of opportunity" and "will not be tolerated". Much of the course outline deals with study skills issues and Mario said that he spends a fair amount of time discussing study skills (and cheating and plagiarism) and manages the course so that students are forced to acquire good work habits. For example, to detect people who [are] off track and force them into it, Mario has his students hand in preliminary design work, which they must then follow. Laboratory preparation and follow-up force students to preview and review concepts. As well, students are allowed to bring one sheet of paper to exams, which avoids the memorization Mario said occurs with closed book exams, but forces students to study and synthesize ideas, which he feels they do not do when preparing for open book exams.

Mario feels that the controversy over the choice of best first programming language is a tempest in a teapot issue. To him, languages should be chosen in terms of how well they can be used to illustrate the concepts of a course. He drew an analogy between teaching programming and teaching physics:

If I were asking how I would teach a physics course, my first question would not be, "What type of oscilloscope should I use in the labs?" I probably wouldn't even say, "Should I use dot notation or prime notation or d notation to describe the derivatives?" I'd probably say, "What kinds of problems should I be investigating? What kinds of analysis should I be showing?" and then the rest of that comes out.

He said that one can make arguments either way in a comparison between Pascal and Modula-2, but that people who argue at that level often are teaching the
language rules rather than teaching people how to use this language effectively to get things going. He uses Scheme in a first course because it has a simple grammar, is highly orthogonal, supports encapsulation and is easy to evaluate. The course could have been taught using PROLOG or Smalltalk or ML (a functional language) or half a dozen languages. ... it probably couldn't have been taught using Pascal or BASIC or ... C. The reason for that is that those languages don't treat procedures as data objects. That's quite necessary to the entire approach that we've taken.

Although he feels that anybody can learn to use tools like Hypercard, Toolbook and so on, Mario feels that not everyone can go beyond that. He cited Alan Kay's analogy of the difference between building a wall and building a Gothic arch. A wall, he said, is easy to build because one can just keep extending it. With a Gothic arch, however, one must plan the whole structure ahead of time and must have a good understanding of the forces involved. To go beyond wall programming, one must be able to create mental models ... to be able to see things at multiple levels of abstraction ... and be able to move quickly from one level to another. Programmers must have certain basic intellectual skills: the ability to solve problems using analytical and synthetic thinking. They must be able to make plans and evaluate their success.

These skills alone will not suffice, however. Mario feels that successful programmers have confidence that they will be able to solve problems and have an attitude that means that they are bothered if something doesn't work. Mario mentioned Seymour Papert's idea of the mathematical way of thinking:

Somehow there are people who look at a problem and don't immediately say "I can't do it." They don't immediately say, "I don't know where to start." They start somewhere and they work their way through until they've got some kind of handle on it and then they go back and try to introspect what's wrong with it.
Mario sees many students in his classes who do not exhibit the traits he feels are necessary for good programmers. These students have such weak problem analysis skills that they are unable to evaluate the requirements of assignments. He wondered at the students who come to him shortly before an assignment is due to ask, What if we didn’t do the project? What if we did something else? He is amazed that students ascribe different meanings to words whose meanings he has stressed in class. He is surprised that weak students tend not to talk to one another and feels that part of the difficulties that students like that suffer ... come from the fact that they’re not communicating anything about ... what’s going on in their head. Weak students, Mario said, have no feeling that the system should be understandable. When something goes wrong, they are concerned not with why it went wrong, but with what they need to do to fix it and will often try random things in an attempt to make bugs disappear. Because these students do not have a well-formed conceptual model and because of lack of confidence, they are afraid to push their model and often become paralyzed when programming.

Mario feels that he can get a pretty good sense for who will be a successful programmer in about 15 minutes. In order to do so he would look at the way that [people talk] about their problem solving skills,

... whether or not they’ve got a real conception of what it means to abstract things out and whether or not they’ve got a conception of how to diagnose the difference between expected results and ... attained results and to determine what to do with it. [He would also look at] the way that they describe how they set objectives for themselves.

**An Integration of Mario’s Knowledge**

Mario seems to have a strong sense of wanting to understand ... things completely on an intuitive level. This quest for understanding is echoed in the joy he gets from
solving hard problems, in the *visceral feel* he has for programs and tools and in the temptation he feels to *push things beyond what you’re supposed to*. In the past, Mario was able to have control of the whole machine. Now, he has a sense of control of whole programs.

Mario’s attempts to understand things completely have given him a broad base of knowledge. He knows many languages and is cognizant of the computer science and education literature. His interest in education as well as computer science is displayed by the fact that he holds a teaching degree and by his interest in the B.C. Computers in Education Committee. He seems able to see many perspectives, to integrate various aspects of an issue and to view issues in a broad context. Thus, for example, he recognizes the fact that professional programmers are strongly shaped by their work environment. He sees students not just as members of a class, but in the broader context of their lives. He recognizes that while teaching programming in elementary school may inculcate traits necessary to educated citizens, the need to teach programming may be subordinate to the need to solve deep social problems. In terms of his own class, he recognizes that some students who seem ill-prepared for his course are in fact not prepared for post-secondary education in general.

Mario attempts to situate his own classes within the broad context of students’ lives. He sets standards of acceptable behaviour. He teaches not just programming, but study skills and analysis techniques. He manages his students’ work habits by having them hand in preliminary design work, by setting up lab requirements that require preview and review and by using exam mechanisms that encourage students to synthesize concepts.

Not only does Mario view teaching programming in a broad context, but he also attempts to integrate salient issues for his students. He teaches computer science as
an organic evolving domain and feels that his teaching is successful when he is able to synthesize for students language, design and analytical issues.

Mario categorized the programming languages he knows in terms of usage. To him, languages are tools with which he can meet various ends depending on his requirements. They can allow him to feel the thrill and personal joy of programming, they can be used to solve problems and they can be the means by which he teaches principles and concepts. Always the underlying principles or uses are more important than implementation details. Thus, when programming, Mario is not frustrated by program bugs. Neither is he concerned with teaching or evaluating program syntax.

When selecting a programming language for a project, Mario will choose one which closely maps the domain of the problem which he is attempting to solve. Because to Mario the solution of the problem is paramount, he is willing to mix languages to achieve the ends he desires.

When teaching programming, Mario is also concerned not with the language itself, but with the ways in which he can apply it. He feels that programming can be used to teach problem solving skills and coping strategies. In his courses, language is secondary to the teaching of object-oriented programming concepts. Thus Mario is not as concerned with the students' understanding of syntax as he is with their understanding of the underlying principles of the style of programming he is teaching.

Mario's comments about the discussion of best first language reiterate his idea that underlying principles are more important than surface details. To him, the first language should be chosen based on the principles of problem solving one is attempting to teach rather than on the syntactical details of specific languages.
Although the use to which a language will be put is important, Mario also recognizes that it is more pleasant to program in a language which fits one's way of thinking. It appears that to him there is an integrated relationship between programmer, language and program. The language structure must have a certain minimum degree of fit, both mentally and to the domain. Given that fit, programmers can shape languages to match their personal styles. The languages that Mario likes and uses most often (Scheme, Smalltalk and C++) fit his way of thinking, allow him to solve the problems he needs to solve and allow him to teach the concepts he wishes to teach.

Mario sees serious tensions in the programming world which seem to emerge from the different needs he has when programming. On one hand, programming gives him a thrill which is increased by the ability to explore the programming environment; to push things. On the other hand, programming is a means of reaching a goal. These two requirements mean that Mario always feels a tension between wanting to explore and better his environment and wanting to get on with the job. This tension is seen in the two types of programmers he identifies: those who want to explore and push the system and those who are intent on reaching a goal. Mario recognizes that the best programmers are those who can combine the two types of personality. Because he thinks that the clever, exploring programmers are the better programmers, though, he seems more apt to bring a sense of discipline to the clever students he sees than to attempt to help the goal-oriented students to become more clever.

Communication skills are important to Mario, both in terms of professional programmers and in terms of students. Programmers must be able to communicate their solutions to others. Thus part of the lab component of his course involves the explication of results. On a broader scale, Mario is concerned about weak students'...
inability to understand course requirements and language and about their ability to be understood. He feels that these students may be suffering because of their tendency to not talk to others about what is going on in their head.

The means by which Mario predicts success in programming highlight the traits that he feels good programmers should have: the ability to communicate, to think abstractly, to set objectives and to evaluate them. Most importantly, he doesn’t think that people will be good programmers unless they have that real sense of wanting to push something to the limits.
Claire

*an instructor, not a programmer*

Claire and I have been colleagues for a year and a half. Last year, she conducted labs in the Pascal course I was teaching and last term taught the course herself. Claire has been working at community colleges for approximately six years, primarily teaching introductory programming courses using Pascal. As well, she has taught introductory computing to students who are not specializing in computing and some second level courses in COBOL, data structures and file structures. Currently, she is teaching BASIC programming to engineering students and is assisting in labs in computer architecture and operating systems concepts courses.

Claire began studying programming because she thought it sounded *interesting* and *new*. Growing up in Mauritius she had no exposure to computing, but had heard about computers and had several friends who had decided to enter the field. Claire obtained a BSc in statistics and computer science in England. Before graduating, she spent a year doing industry research, working in BCPL and FORTRAN. She received her Master's degree in computer science in Calgary and then began teaching in the college system. As Claire told me, she has never worked as a programmer to earn money.

Claire has used and taught only procedural languages: PL/I, FORTRAN, COBOL, C, ALGOL, BCPL, assembly language and BASIC. She has had some exposure to Modula-2 (a procedural language) and LISP (a functional language), but, as she said, has never had a chance to teach them so does not know them very well.

Several times during the course of our interviews Claire asked my opinion about questions I had asked her and often said, *That's what I think. I don't know.* Her
responses often ended with the phrase, isn’t it? and often began with a restatement of my question. For example:

Debbe: If you had infinite resources, ... would you teach it differently?
Claire: The same course?
Debbe: Yeah. Yeah.
Claire: Would I teach it differently? Mmm ...

Claire’s Knowledge about Programming

Claire decided to study computing because it seemed to her to be interesting and because friends were studying it, but she did not know what [she] was doing; she did not understand what programming entailed. Her decision to enter the field was made in a way different from most people’s, she feels. In Canada today, she said, people have exposure to computing and may have relatives working in the field. They therefore base their decisions on an understanding of computing that she did not have when she began her studies. At first Claire found her courses somewhat difficult because she didn’t know what to expect and because she was making an adjustment to life in England, but things became easier as she made friends and received instructors’ help. She said that her interest in computing just grew. She decided to concentrate on computing because it was more interesting than mathematics. She was thinking in terms of jobs ... and also, it was the thing to do. Many of her friends were working in computing and there was more interest in that field.

To Claire, programming means writing code. It is separate from problem solving and designing solutions. Claire said that she likes problem solving, but that she doesn’t like programming. She likes the stages ... you go through when solving a
problem. The languages -- well, yeah, of course, you know, sometimes you like this language, you like the other one, but I think [what is] more interesting is the problem solving part, don't you think? Debugging is also part of the programming process. When we discussed the possibility of teaching programming to elementary school students, Claire revealed some of her thinking about programming. It appears that to her programming requires the ability to read and spell. While pondering the benefits of teaching programming to kindergarten students she said, Can they be taught to write a simple program? To add two numbers or something like that?

When she mentioned programs she had written or projects she had worked on, Claire invariably outlined implementation details, rather than program functionality. I asked about the topic of her Master's thesis and she replied that it dealt with graphics and database and linking between the two databases. The industry research she did involved [looking] at some modules and ... [making] some changes and [adding] on ... some modules. The last program she wrote -- one that she had assigned to students -- was about pointers and used linked lists and queues and stacks.

Claire categorized languages primarily in terms of language structure and syntax. Although she mentioned that COBOL is used for business applications, she also sees similarities between it and PL/I in terms of data declaration syntax. PL/I is also like Pascal: when you program, it's a little bit like Pascal as well, you know the langua -- the syntax of the language? Pascal, to Claire, is similar to both ALGOL and C. She found Pascal easy to learn once she had learned ALGOL, because the languages are very similar ... structure-wise. Pascal is like C because what you can do in Pascal, ... you can do that in C -- ... loops, ... pointers, ... if statements, ... selection, branching and all, ... you can still do that in C .... I think of C as just being another language.
Although Pascal and C are similar, Claire also grouped C with assembly language and BCPL, because all three languages allow the programmer to get close to the system.

To Claire, LISP is very different from the other languages with which she is familiar.

Claire:  When I look at LISP, ... it's a language ... for list processing more and ... it looks, when I look at it I say, oh, that's LISP ... it's so different from the other ones.

Debbe:  Yeah.

Claire:  You know, all these brackets.

Debbe:  OK. So the way it looks is different.

Claire:  Yeah -- the syntax.

Claire placed both BASIC and FORTRAN in categories of their own: FORTRAN because it is a scientific language and BASIC because it is easy to learn.

Claire's favourite languages are Pascal and C. She likes C because you can get a little bit closer to the system. ... it allows you to do new things. Pascal is good ... as a first language in an educational environment. COBOL is Claire's least favourite language, although she found it easy to learn. Its syntax is very different from Pascal's in that it consists of a lot of words. The structure is different, isn't it?, she said. The sections, paragraphs and wordiness of COBOL, she thinks, make it different from other languages.

Claire said that when she is programming she tends to think in terms of [Pascal] structure, perhaps because of her extensive experience teaching Pascal. To her it does not feel different to code in different languages. When I asked her to tell me how she would code in C a process to find the length of a string, her response was very procedural: In a loop you would check ... until you find ... you test for that null
character, so if you haven’t found it, you increment, keep going. Claire finds the BASIC that she is teaching limited because it has no while, else or nested if constructs. She said that she is not judging BASIC based on the capabilities of Pascal however, but based on previous versions of BASIC that she has learned.

To Claire, a good programmer is somebody who can think and write code; somebody who ... has some methodology, ... knows ... how to go about doing things in a systematic order, who can actually translate ... the ideas into code. ... and somebody who knows how to communicate with people as well. ... it’s not just writing code. A programmer sits in front of a computer all day surrounded by computer printouts. She said, his job is to write programs to get something to work. Am I right? I don’t know.

Claire feels that there are two distinct types of programmers: systems programmers and applications programmers. One of her brothers is a systems programmer and one an applications programmer. To her, systems programmers have the more interesting job. She said that being a systems programmer would be like being a grad student in that one would be constantly learning new things. The systems programmer makes sure that everything is running smoothly and is like a doctor who must diagnose the system. Application programmers, on the other hand, are more concerned with programming languages and what code they are going to write. They get specifications and instructions from higher ups, they do maintenance, fix bugs and test code using languages like COBOL to write applications like accounts receivable and accounts payable. Their code deals with printing reports, updating files and prices of products. Although she says that her brother likes applications programming, she also mentioned that that’s the job that he gets ... because he can’t get any better.
Claire has wanted to teach since she was in university. She laughed as she said, *I remember after graduation I told everybody, “I’m going to become a teacher”*. Now she says that when people ask, *I don’t say I’m a computer programmer. I say I’m an instructor*. Claire likes teaching. *I don’t know why I teach*, she said, *but I like teaching*. She likes being with people and seeing students make progress. She receives satisfaction from seeing them succeed in understanding an idea or finding a bug with her help. She likes to hear them say, *oh, thank you [Claire] ... this is easy*. She also likes seeing students after they have finished her classes. She said, *Now I get to see students from last year. You know, the ones that were doing Pascal ...?* and named several students who were in our class last year. Claire feels that she has the patience necessary to be a good teacher. *I can teach the same thing many many times to the same person*, she said.

Claire likes preparing for classes, particularly for ones which she has not taught before. *Although it’s very challenging and could also be scary*, she likes the preparation because it gives her an opportunity to learn new things or things that she has forgotten. Despite the fact that Claire does not like the volume of marking that she does, she feels that it is good feedback for her and for the students. Seeing how they are doing, she said, is *how they will learn*. Claire also does not like having to *put up with a student like A*, a student who has, she said, a complaining attitude and who makes her feel *stupid in front of the whole class*. She feels that if she were older, she would be better able to deal with this student, who is *many years older* than she is.

Claire appears to be very concerned about students. When I asked her to explain some code to me as though I were a student, she spent a long time asking about the
‘student’s’ background and understanding before she explained the code. She said that the explanation would depend on what the student was comfortable with: So, if it’s the language that’s a problem, so then maybe I would explain ... the language. If it’s not, it’s just ... what the program does ... overall. Claire is also aware of students’ concerns. She said that it would be better if the students’ workload were not so heavy so that they would have more time to devote to programming.

Claire’s outline for the Pascal course is the same one that I used, with very minor changes. If more time were allotted for the course, she said, she would introduce additional topics, like graphics, because people ... tend to like ... pictures on the screen and they like to know how to [create them].

Although she is unsure, Claire feels that it may be a good idea to teach programming to elementary school students in order to expose them to the technology and its applications and to the idea that programs are required to make computers work. Young children are interested in computers, she said, and many of them could learn how to program. Because the people that she teaches have already been exposed to computers and have some programming experience though, she feels that the purpose for which she teaches is different.

Claire:  ... the purpose would be ... to still teach them how to write good programs ... how to solve a problem and how to ... from there write a good program.

Debbe:  And for what purpose? Why would you be teaching them to write programs?

Claire:  I don’t know. If they want to solve a problem ... using the computer, OK, so to show them ... the way to do it.

Claire mentioned that she is not sure why her engineering students are learning BASIC on their hand-held computers: well, maybe because in case they have to ...
solve a problem in BASIC so you can just do it right away. Claire agreed with my suggestion that perhaps the students would need to use programs on the job.

Claire appears to see programs in terms of their modular structure. When explaining code to me she first explained the modular structure, drawing a diagram to illustrate the program hierarchy, and then explained each function and procedure call in detail. As she described the workings of each line of code, she also explained the theory behind it. For example, when explaining a particular function she said she would step through this and explain what this is doing ... and how it relates ... to the complete rest of the program. ... that's a function ... now at this point [I would] explain again what's the difference between a function and a procedure. As she went through the program, she also stressed the importance of internal documentation and meaningful variable names.

When I asked if she tries to impart a particular image to her students, Claire replied that she reminds them that they have to become good programmers; that they have to have this attitude. In particular, programmers must have patience when solving a problem or finding a bug. Claire feels that students have particular problems with specific structured data types. Pointers are difficult, she said, and the students get lost. Claire draws diagrams to help the students understand pointers and data structures such as trees that are based on them. As well, a lot of students find [arrays] difficult to understand at first ... it is too abstract to them or something.

Pascal is Claire's favourite language to teach and she thinks it is the best language to teach first to students. Pascal is easy to learn and is a good introduction to structures and modular programs. It is a good tool for teaching programming techniques and concepts. Although she likes Pascal, Claire would like to have an opportunity to teach C as the first language at the college level, particularly because many of her
students already know BASIC or Pascal. She does not know whether C would be better, but said, *maybe we should try and teach C and see and compare.*

Claire feels that programming is *like any other subject* in the sense that some students succeed at it and some do not. Anybody with motivation and interest can learn how to program, but some are better at it than others. Some are *be able to* ... *think ... more quickly or have better ideas when it comes to designing programs.* Success in programming does not depend on age or computer exposure. However, Claire feels that very young children, because they cannot read or spell or add, cannot learn to program. Claire thinks that other factors, such as being away from home or returning to school after many years away, will affect how well students succeed in post-secondary courses. She does not feel that she can predict students' success, but judges how well they are doing based on feedback she receives in the first two or three weeks of class. She measures their progress by marking their assignments, by looking at the work they do in labs and by listening to the questions they ask to see whether *those questions ... should have answers that they should already know.*

**An Integration of Claire's Knowledge**

Claire has never worked as a professional programmer. Her educational experience seems to shape the way she thinks about programming. Claire likes learning. She said that she enjoys preparing for new courses because it allows her to learn new things. As well, she is drawn to systems programming in part because it is *like being a grad student.* Systems programming also involves problem solving, which is what Claire likes about programming. She dislikes writing program code, which, in her perception, is what an applications programmer is most concerned with.
To Claire, programming is *like any other subject* in that anyone can learn how to do it. Some students will be quicker and able to think of better program design, but Claire’s sense of programming as a subject to be learned appears to be echoed in her impression that anyone who is motivated can learn it. Claire’s ability to predict who will be a good programmer is based solely on their performance in her classes. Her sense of programming as a subject is further evidenced by her feelings about why it should be taught. She does not have a strong sense of the purposes to which students might put their programming knowledge. Rather she sees programming being taught *in case* [students] *have to solve a problem ... using the computer.* Neither does Claire explain her own programming experience in terms of its purpose, but rather in terms of language implementation details.

Claire concentrates on the syntax and structure of programming languages. She has primarily taught Pascal and all the languages with which she is familiar are procedural. Claire’s coding and teaching make evident this background. She said that she tends to code in a Pascal style and during our interview her explanations and descriptions were strongly procedural. Claire categorizes languages by their structure and syntax and judges them based on these attributes. Her sense of programming as a very syntax-based activity is echoed in her feeling that young children cannot be taught to program because they cannot read or spell. Furthermore, her idea of a simple program that they might be asked to code appears to reflect her sense of what programming is about: *Can they be taught to write a simple program? To add two numbers or something like that?*

Claire sees programs in terms of their modular structure -- she describes programs by drawing Module Hierarchy Diagrams and explains them in a top-down fashion, moving from main logic to details of subroutines and then to specific lines of code. Good documentation is also important to Claire. Thus she takes an expert approach
to teaching programming. Pascal is a good first language because it allows Claire to teach the aspects of programming that are important to her: structure and modularity. Typical student problems that Claire cites relate to data structures or to overall program hierarchy. Her musings about teaching C seem to highlight a curiosity based on the sense she has that C is just another language.

To Claire, programming is writing code. Thus a good programmer is one who can translate ideas into code. As well, a good programmer does things in a systematic order. Claire feels that both good programmers and good teachers need patience and that an important trait of a good programmer is the ability to communicate well with others.

Interaction with other people seems important to Claire. She learned programming because others had done so. She found it hard to adjust to university, but other students and teachers made the transition easier. Her decision to concentrate on computing rather than on math was made partly because of the influence of her friends. Claire said that she might think differently about programming if she were exposed to different groups of students. The influence that others have on Claire is evident in her language, which is filled with you know and isn’t it? and in the fact that despite having taught Pascal many times in the past, she did not make changes in the course outline which had been used the year before.

Claire likes teaching because she likes being with people. She enjoys seeing students from previous classes and is affected by student attitudes. Students’ needs and desires are important to Claire. She feels satisfaction when she sees them succeed and recognizes that outside factors influence that success. She is concerned about determining what students’ background and levels of understanding are. She
dislikes marking, but recognizes that it is useful to students. As well, she would change her course, if she could, to add topics, such as graphics, that students enjoy.

Claire's perception of programming appears to be strongly affected by her educational experience and by the fact that she teaches mainly first term courses. She seems to judge programming and programming languages based on the requirements of first term students. Her perspectives make it clear that Claire is an instructor, not a programmer.
CHAPTER FIVE

DISCUSSION OF THE RESEARCH RESULTS

Introduction

That qualitative research involves hypothesis generation rather than hypothesis testing (Spector, 1984; Jacob, 1988; Lincoln and Guba, 1985; Glaser and Strauss, 1967) became apparent as this study unfolded. At the outset, the goal of the study was to investigate two questions: How do teachers organize their knowledge about programming? and How do teachers' constructions about programming integrate with their constructions about teaching programming? As the data was collected and the analysis began, though, it became clear that it was impossible to investigate the second question, which deals with integration, without also investigating the ways in which teachers organize their knowledge about teaching programming. Thus a third research question: How do teachers organize their knowledge about teaching programming? was generated and the results of the interviews were reported in the form of responses to three research questions.

As the results of the interviews were described, however, it became difficult to separate the respondents' perceptions of programming from their perceptions of teaching programming. Issues described under the heading “Knowledge about Programming” (programmers' attributes, for example) reappeared under the heading “Knowledge about Teaching Programming”. The two knowledge domains seem to be intertwined and it appears that they are constructed within the same conceptual frameworks. Furthermore, although in some instances it was tempting to infer causality in the relationships among emerging themes, for the most part the themes were interconnected; the links of influence among them multi-directional.

What Teachers Know about Programming

What Teachers Know about Programming 146
When it came time to analyse patterns across respondents, I had a strong sense that to separate the analysis into two sections -- one an investigation of knowledge about programming and the other an investigation of knowledge about teaching programming -- would be artificial and detrimental. As Stake (1978) says, "truth in the fields of human affairs" (p. 6) is approximated by using "statements that are rich with the sense of human encounter" (p. 6). The results of this study indicate that in order to arrive at an approximation of "truth" in the field of computer education, teachers' constructions about programming and about teaching programming must be examined within the same frameworks. It appears that the question is not How do teachers' constructions about programming integrate with their constructions about teaching programming?, but rather, Within what frameworks do teachers construct their knowledge about programming and teaching programming?

This chapter examines the data from this more integrated perspective by means of a discussion of five conceptual frameworks within which the respondents appear to construct their knowledge about programming and teaching programming:

1. life experience,
2. social context,
3. desire for control,
4. paradigm experience, and
5. mental programming models.

Conclusions drawn from the discussion are then summarized and the limitations and implications of the study are outlined.
Conceptual Frameworks

Life Experience

The respondents' academic and industry backgrounds are varied. Table 3 displays their life experience and their ideas about the reasons for which programming is taught.

Table 3

Respondents' Life Experience

<table>
<thead>
<tr>
<th>Respondent</th>
<th>Work Experience</th>
<th>Academic Training</th>
<th>Teaching Experience</th>
<th>How Programming was Learned</th>
<th>Why Programming is Taught</th>
</tr>
</thead>
<tbody>
<tr>
<td>Melissa</td>
<td>&lt; 5 years</td>
<td>college, university</td>
<td>&gt; 10 years</td>
<td>academic, self-taught</td>
<td></td>
</tr>
<tr>
<td>Claire</td>
<td>none</td>
<td>post-graduate computing</td>
<td>6 years</td>
<td>academic</td>
<td>as a subject</td>
</tr>
<tr>
<td>Isaac</td>
<td>&gt; 5 years</td>
<td>part-time college</td>
<td>&gt; 10 years</td>
<td>self-taught</td>
<td>job training</td>
</tr>
<tr>
<td>James</td>
<td>&gt; 5 years</td>
<td>post-graduate</td>
<td>&gt; 10 years</td>
<td>self-taught</td>
<td>job training</td>
</tr>
<tr>
<td>Lorne</td>
<td>&gt; 5 years</td>
<td>no computing, some teacher training</td>
<td>&gt; 10 years, part-time</td>
<td>self-taught</td>
<td>job training</td>
</tr>
<tr>
<td>Kyle</td>
<td>&gt; 5 years</td>
<td>college, university</td>
<td>&gt; 10 years, part-time</td>
<td>academic</td>
<td>job training</td>
</tr>
<tr>
<td>Mario</td>
<td>&lt; 5 years</td>
<td>post-graduate computing, teacher training</td>
<td>&gt; 10 years</td>
<td>academic</td>
<td>liberal art</td>
</tr>
<tr>
<td>Ross</td>
<td>&gt; 5 years</td>
<td>post-graduate computing</td>
<td>&gt; 10 years</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Two of the respondents, James and Lorne, learned programming solely in the workplace. Isaac too has a strong industry background, but has also taken many computing courses on a part-time basis. The other respondents have formal computer training, either at university or college. Melissa and Kyle studied at both college and university and Claire, Mario and Ross have done post-graduate work. Claire alone has no industry experience. She also has the least amount of teaching experience of the respondents. Claire has been teaching for six years, whereas each of the others has been teaching for over ten years. All of the respondents but Kyle and Lorne are full-time teachers. Mario, Melissa, Lorne and Claire appear to have a strong interest in teaching as a profession. Claire said that she has always wanted to teach and Melissa reminded me that she teaches everything [she learns] how to do. Both Mario and Lorne have professional teacher training and Mario taught for some time in the elementary school system.

The respondents' experiences form a framework within which they construct their knowledge about programming languages and the ways in which students learn them. Claire's strong academic background is evident in her perception of herself as an instructor, not a programmer. She views programming and languages within the context of students' requirements. To her, programming is a subject. James' and Lorne's extensive industry experience contributes strongly to their perceptions. Both categorize languages in terms of the applications for which they are used and both feel that students learn by experience. As teachers, they act as colleagues to their students, preferring the discovery mode of learning to the lecture mode. Furthermore, Lorne's emphasis on students' need for human, computer-based and print-based resources when learning computing echoes the way in which he learned to program. Isaac, who learned programming in the trenches, feels strongly that students learn programming through practice, practice, practice. Nonetheless, Isaac,
along with Melissa, Claire, Kyle and Mario, uses lecture mode when teaching. Since
this is the standard mode of instruction at their schools, though, it is not possible to
judge whether or not this is a personal preference.

Industry and academic background appears to have contributed greatly to the
respondents' views about the purpose of teaching programming. Claire, whose
exposure to programming is purely academic, and who learned programming
because it was the thing to do, seems to feel that programming should be taught
solely in the event that students have to solve a problem ... using the computer.
She did not offer examples of ways in which they might apply their programming
knowledge. Isaac, Lorne and Kyle, who have extensive industry experience, all
suggested that they teach programming for industry training. James, who learned
programming in order to solve problems on the job, teaches programming as a
means to an end -- not for the training of programmers per se, but in order to help
analysts solve problems. Mario's teaching education and experience appear to shape
his view that programming should be taught as a liberal art.

The recency of the respondents' experiences also contributes to their perceptions of
languages. Those that programmed some time ago in a language with which they
no longer work seem to have what Mario called a mental model of the language
that may not match its current implementation. Thus Lorne's and Melissa's views
of BASIC, Mario's view of FORTRAN and Ross's view of COBOL are based on their
views of the languages as they were implemented in the past and may be what
Mario called emotional reactions to languages based on dated perceptions.

Several of the respondents expressed views about the future of the computer
industry which appear to be shaped by their experiences with that industry.
Although Isaac sees the need for fewer programmers in the future because
development systems are becoming more intelligent, he feels that most industry programming is still done in the procedural style. Both Lorne and Kyle feel that the industry is shifting toward an object-oriented paradigm whereas Ross feels that languages are [evolving] more and more towards a LISP-like model. These differing views of the industry are reflected in the respondents' views about choice of teaching language. Isaac teaches procedural languages, Lorne and Kyle teach Smalltalk, an object-oriented language, and Ross teaches LISP. Each of the four expressed the opinion that students' professional careers would benefit from knowledge of the language he teaches.

Thus the respondents' academic and industry experiences help shape their views about the programming industry, programming languages, the ways in which students learn programming and the reasons for which it should be taught.

Social Context

The respondents differ in terms of the social contexts within which they framed their interview responses. The views of Kyle, Mario and Claire, in particular, appear to be strongly linked to personal, interpersonal and societal concerns. The other respondents did not express strong views in these areas.

Much of Kyle's discussion was framed in the context of the importance of achieving a balance emotionally, mentally, and spiritually in life. This quest for balance appears to contribute to his view that programming is just one aspect of life, which sometimes threatens to outweigh other aspects. It informs his description of programming as an engenderment of creativity, practicality, intellectual stimulation and aesthetic beauty, an image which he attempts to convey to his students. As well, the broad context within which he views people's lives shapes his sense that
programmers' attributes are just one aspect of their characters and that his course is just one aspect of students' workload.

Mario also appears to view programming and teaching within a broad context, although his perspective appears to be more societal than Kyle's. He teaches not programming, but general principles about problem solving that an educated citizen in our society ought to have. His course [interacts] with the rest of what makes [students] competent and is presented in such a way as to establish social and educational behavioural expectations. He views people within a broad social context: recognizing that professional programmers are shaped by the environment in which they work; realizing that students who have trouble in his class may be ill-prepared for university in general; and questioning the importance of teaching programming in elementary schools when students have serious social problems with which to contend. As well, the reasons for which Mario selects languages with which to perform specific tasks are framed within the context of the environment in which programs will be used and maintained.

Claire's views appear to be framed within the context of personal interaction. She is aware of the broad context of students' concerns, attitudes and desires. As well, her impressions of different kinds of programmers are shaped by her perception of her brothers' jobs. Claire's concentration on teaching first year courses and the fact that she has never programmed professionally seem to have strongly shaped her perceptions about programming in general. She said that she might think differently about programming if she were exposed to different groups of students. As it is, she categorizes programs as a first year student might: in terms of syntax and structure.
Thus it appears that for some programming teachers, discussions of programming and teaching programming are firmly situated within a social context.

Desire for Control

The ways in which the respondents deal with control appear to shape the ways in which they view programming and the teaching of programming. Of all the respondents, Melissa seems to have the strongest desire for control. She likes programming, as she likes flying and motorcycle riding, because it gives her control of machinery. Her favourite language is assembly language because it gives her a feeling of control over low-level machine functions. That to Melissa programming is a means of controlling is further demonstrated by her sense that computing can be demystified by giving students a degree of control through programming. Teaching also gives her a sense of control; She enjoys having people listen to her. As well, Melissa’s desire for control shapes the way in which she teaches. She feels that teachers have a strong influence over students. Melissa’s students submit preliminary design work so that she can streamline their logic. She feels that teachers can influence students’ feelings about programming languages by displaying their personal biases about particular languages.

To a lesser degree, Mario, Ross, Kyle and James are also concerned with control. Mario said that when he first began programming it was very easy to write a program that would take over the whole machine. He liked the sense of control he had when he could push things to their limits. Now he likes to feel that he has control over his programming environment. Furthermore, he feels that good programmers are those who want to explore the boundaries of the system and control their working environment. In his courses, Mario goes beyond the teaching of programming by attempting to give his students not only educational, but also
ethical guidelines. His course outline clearly delineates the boundaries within which he expects his students to behave.

Ross also spoke about the sense of control programming gives him. He said that whereas he used to enjoy having control at the assembly language level, he now wants to exert control by changing the behaviour of the inference engine. When using LISP, he said, he is not a slave to the tool, as he was when using COBOL and other block-structured languages. He dislikes Smalltalk and PROLOG in part because he cannot control them to the extent that he would like. In the classroom, Ross exerts a subtle kind of control. He guides students into LISP-world by pulling strings and pushing buttons and teaches students his techniques by showing them how to give prospective employers little tempting nibbles and then withhold.

Through programming Kyle feels that he can manipulate the world. This sense of control over malleable software objects, Kyle said, sometimes spills over into the world of physical objects and interpersonal relationships. Kyle contends that attempts to control people in the way that he controls software objects represent a psychological defect which he tries to rise above. Nonetheless, Kyle has strong opinions about students' motivations and the ways in which learning takes place and appears to feel that in some cases he should be able to influence others to do things the right way. For example, he says that the conditioning that causes some students to be inhibited about learning in an active way should ideally be broken.

James categorizes languages partially in terms of their power and likes C and C++ because they are powerful languages. When programming, James enjoys a feeling of power over [the] machine, but also says that programming allows him to tap into the power of the computer. Thus there is a sense of a sharing of power. This
complementary relationship is echoed in the classroom as James works together in the lab with his students, trying out new things depending on what happens.

Isaac is very concerned with discipline. Programmers need to be disciplined, he feels, and his favourite languages are those that impose discipline on the programmer. Although he says that programming teachers are just guides, Isaac imposes strict discipline on his students. He forces students to write code in a particular style and attempts to constrain them through his project specifications. As well, in his note to me he expressed the feeling that structured programming style "must be hammered repeatedly". It appears that for Isaac, programming is best done when the programmer does not have control over every aspect of the programming environment, but rather when the programmer is constrained by the environment. In the same way, learning happens best when students are constrained by the teacher.

The ways in which the respondents exert control and are controlled, then, appear to play important roles in shaping their programming and teaching styles.

**Paradigm Experience**

Although each of the respondents' first language was procedural, they vary markedly in terms of the breadth of their experience with other programming paradigms. Figure 4 displays the language paradigms with which each respondent is familiar in a form that gives some sense of the breadth of their knowledge.
### Figure 4. Breadth of Respondents’ Paradigm Experience

<table>
<thead>
<tr>
<th>Respondent</th>
<th>Logic</th>
<th>Functional</th>
<th>Procedural</th>
<th>Object-Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Melissa</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Claire</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Isaac</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>James</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lorne</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kyle</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mario</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ross</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As Isaac pointed out when I asked him about the languages he knows, *there is 'know' and there is 'know'*. In Figure 4, solid rectangles mean that the respondents ‘know’ a paradigm in the sense that they have worked with it and feel that they understand its attributes. Shaded rectangles mean that the respondents have had some exposure to a paradigm, but do not claim a strong understanding of it.

Melissa’s and Claire’s experience has been procedural, although Claire has had some exposure to LISP. Isaac too has programmed extensively in the procedural paradigm, with a brief introduction to Logic and Functional programming with PROLOG and LISP and some exposure to the object-oriented languages Smalltalk and C++. James, with a procedural background and some exposure to PROLOG, now uses C++, in the object-oriented paradigm, extensively. The remaining respondents, Lorne, Kyle, Mario and Ross, have experience in all four paradigms.

*What Teachers Know about Programming* 156
The breadth of the respondents' paradigm experience appears to contribute strongly to the perspective from which they view programming. Melissa, who knows only procedural languages, said that one procedural language is ... like another. To her, programs are linear and sequential in nature and she differentiates between them in terms of level of detail. Similarly, Claire's discussion of language categories was notable for the similarities rather than for the differences she perceived between languages. To her, LISP is different from other languages not because it is based in a different paradigm, but because of its syntax. Isaac, who has limited experience with non-procedural languages, differentiates between paradigms, but his discussion of procedural languages was much more specific and detailed than his discussion of the other paradigms. James, who has recently shifted to the object-oriented paradigm, did not categorize languages on the basis of paradigm. However, he sees a strong difference between object-oriented programming and procedural programming. Lorne, Kyle, Mario and Ross, in contrast to the other respondents, were able to categorize languages within the broad context of the paradigms with which they are familiar. All of them compared and contrasted the various language paradigms and Kyle even added two paradigms of his own definition.

All respondents but Melissa and Claire feel that programmers favour languages that fit the way they think. As Isaac mentioned, one is able to choose a language that fits one's way of thinking only if one is good at many different languages. Lorne, Kyle, Mario and Ross seem to be able to choose their languages, each preferring to program in and teach a paradigm with which he is comfortable. It appears that because of their broad base of experience each is able to select particular languages for particular tasks. Furthermore, Lorne and Mario described the ways in which they sometimes mix languages from disparate paradigms. Ross, Kyle and Mario mentioned not liking to work with certain languages that they have to fight; that get
in the way. The other respondents are more restricted in their choices. Melissa, Claire and Isaac teach the procedural paradigm, with which they are familiar, and James encourages his students to use C++, although they are free to use either C++, an object-oriented language, or C, a procedural language, in his courses.

Almost all of the respondents use an expert approach to teaching programming, in the sense that they expect their students to solve programs in the way that experts do. Because they are teaching various paradigms, though, the expert approach that they are using may differ from the classic definition, which involves the use of the ‘stepwise refinement’ model. Melissa, Claire and Isaac, strong procedural programmers, do expect their students to use top-down structured programming methods. Kyle, James and Ross, on the other hand, expect students to use the methods that an expert object-oriented programmer or an expert functional programmer would use.

All of the respondents feel that programming in a particular language shapes one’s way of thinking. Melissa, Lorne, Kyle and Ross each gave examples of situations in which they or their students wrote code based on the style of another language. Kyle described forcing a procedural solution into an object-oriented language and Ross remembered how design methodologies spilled over from the [procedural] paradigm to LISP. Melissa stressed that, although both are procedural languages, to program in PL/I students must lose the old COBOL restrictions and Lorne described having to make his thinking change to fit a new language. Isaac might have been speaking for all the respondents when he described the sense he has about the way in which languages shape our thoughts:

*We and hundreds of others like us have trained our minds to think of problem solutions based in a certain way of doing things, so suddenly having to solve a problem in a way that’s foreign to*
us is going to be very difficult. Which obviously lends strength to 'we think in the way that our languages deal with problems'.

Because of their strong sense that languages shape the way in which programmers think, the respondents were very adamant about the importance of a student's first language. In particular, the respondents that have extensive experience in more than one paradigm pointed out the difficulty involved in shifting from one paradigm to another. They see the paradigms as very different and feel that the shift requires a radical change of thinking, as Lorne said.

The respondents themselves found the shift hard. Several of them spoke about their tendency to drop out of the new paradigm back to the procedural way of thinking. They feel that students find the shift difficult also. Ross said that he sees students being dropped rather cold into a shocking new paradigm. Kyle spoke about students being tainted by the limitations of procedural languages and James said that teaching procedural languages first puts blinkers on the students. Both Ross and Kyle stated that when teaching students they try to expand their horizons in terms of language and uses of computers in general.

The respondents use various techniques to make the shift easier for their students. Kyle tries to avoid overwhelming students with too many new concepts by making his first assignment a familiar procedural problem that the students solve in an object-oriented way. Lorne suggested the use of C++, an object-oriented extension of a procedural language, as a gradual way to introduce the paradigm shift, which would allow the teacher to be subtle about the changeover. Ross gives his students tempting little tasks ... where it would feel artificial to program in the style of the procedural paradigm. Although the respondents feel that programming in the procedural paradigm is vastly different from programming in other paradigms, Ross alone expressed the opinion that different students do well in different styles of
programming. As he said, the students who catch on to LISP are a much different category of programmers from those who do not.

It appears, then, that the breadth of the respondents' paradigm experience shapes the ways in which they perceive and teach languages. Those with broader experience choose and teach languages that fit their thinking styles. The importance of the first language, recognized by all respondents, is felt keenly by those with broader experience, who are attempting to educe from students a shift to a new paradigm.

**Level of Mental Model**

During our discussions, Kyle and Mario alluded to the mental or conceptual models of programming being used by programmers. Melissa, James, Mario and Lorne pointed out that programmers need to be able to think at varying levels of abstraction. As James put it, programming requires a strange mix ... of someone who thinks broadly and at the same time can implement those broad thoughts through the details of a program. Although it appears that programmers, as Mario said, must be able to see things at multiple levels of abstraction, each respondent tends to concentrate on a particular level. It is possible to place the respondents on a scale that represents the level of abstraction of their mental models of programming.

The scale in Figure 5 ranks programmers' mental models from the concrete to the abstract. At the concrete end of the scale is a model based on the machine itself. At the abstract end is a model based on the programmer's imagination. Between them lie models based on the structure of programs, the programming environment (i.e., the tools and systems that interact to form the programmer's 'workbench'), the physical world and a model based on a mental conceptualization of the physical world. A model of the physical world differs from a model based on a
conceptualization of the physical world in the following sense: respondents placed on the scale at the Physical World position stated that when programming they are constructing models of the world. Those respondents placed on the scale at the Conceptual World position talked about modeling people's *conceptual models* of the physical world.

Respondent

Melissa

Claire

Isaac

James

Lorne

Kyle

Mario

Ross

Machine Program Structure Programming Environment Physical World Conceptual World Imagination

Concrete

Abstract

**Figure 5. Respondents' mental models**

Respondents' locations on the scale are represented by dark rectangles. These rectangles position the respondents in terms of the entity they appear to be modeling when they are writing a program. This position represents a tendency only since most of the respondents appear able to create models at different levels. A black line represents the range of models that each respondent tends to use.
Respondents were placed on the scale based on the way in which they talked about programming. The metaphors they used when describing programming often offered useful insights into their conceptualizations. Melissa, for example, said that when programming, she picture[s] things physically happening as a result of program instructions. Her metaphors were very physical and were accompanied by distinct hand motions. Although Melissa seems typically to create mental representations of the physical machine when she is programming, she also appears to have a mental model of program structure. For example, she talked about getting a building block of code to plug into her program.

Thus Figure 5 illustrates that Melissa typically models the physical machine, but also creates mental representations of program structure. It appears that Claire typically models the structure of the program that she is writing. To her, programming is writing code. Her descriptions of the programs on which she has worked were descriptions not of function, but of implementation details. Isaac seems to concentrate on the level of the programming environment. To him, programming involves working with complex systems in which many of the parts interact, assimilating a large number of details and tying all the pieces together. Although Isaac typically concentrates on the programming environment, he also at times showed evidence of thinking about both real world problems and the more concrete aspects of programming -- the program structure and the physical machine.

James and Lorne, it seems, tend to concentrate on modeling the physical world. To James, physical world models are important, not code. He appears as well to have strong models of programming environments and program structure as evidenced by the way in which he explained code to me first in terms of its broad structure and then in terms of specific details. Lorne said that using the object-oriented approach allows him to think about real world objects, their attributes and behaviours. He
also appears concerned, however, with more concrete aspects of programming. The picture he drew to illustrate a Smalltalk program shows his ability to represent program structure and the programming environment. As well, he feels strongly that programmers need an understanding of low-level machine details.

Kyle and Mario appear to model not just the physical world, but human conceptualizations of the physical world. Kyle said that the object-oriented paradigm reflects the ways in which people think about the world. Mario models the principles or concepts that he wishes to teach or investigate. He also talked about modeling the problem space. Whereas Kyle is not interested in models of a more concrete nature, Mario appears to use many levels of model, from the structure of a program to the physical world. He talked about having a visceral feel for programs and the programming environment.

Ross also seems to be modeling a mental construction, but where Kyle and Mario are modeling conceptions based on the physical world, Ross is modeling conceptions that come out of the air; out of his imagination. He models at more concrete levels as well. Although he used to be interested in low-level machine details, he is now more interested in modeling the LISP environment.

Table 4 compares the respondents’ preferred mental models to their views regarding aspects of programming and teaching programming.
Table 4

The Relationship of Mental Model to Respondents' Views

<table>
<thead>
<tr>
<th>Respondent</th>
<th>Mental Model</th>
<th>Preferred Paradigm</th>
<th>Teaching Models</th>
<th>How Languages are Categorized</th>
<th>Programmer Attributes</th>
<th>Best First Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Melissa</td>
<td>machine</td>
<td>procedural</td>
<td>machine</td>
<td>ease of manipulation</td>
<td>logical, patient, can see details</td>
<td>assembly</td>
</tr>
<tr>
<td>Claire</td>
<td>program structure</td>
<td>procedural</td>
<td>program structure</td>
<td>structure and syntax</td>
<td>good coder, patient, systematic</td>
<td>Pascal</td>
</tr>
<tr>
<td>Isaac</td>
<td>programming</td>
<td>procedural</td>
<td>details and patterns</td>
<td>paradigm</td>
<td>can grasp detail, creative</td>
<td>Pascal</td>
</tr>
<tr>
<td>James</td>
<td>physical world</td>
<td>object-oriented</td>
<td>varying levels</td>
<td>use</td>
<td>independent, can see big picture</td>
<td></td>
</tr>
<tr>
<td>Lorne</td>
<td>physical world</td>
<td>object-oriented</td>
<td>physical world and program details</td>
<td>use</td>
<td>can break system down, can see multi-levels, creative</td>
<td>assembly</td>
</tr>
<tr>
<td>Kyle</td>
<td>conceptual world</td>
<td>object-oriented, AI</td>
<td>paradigm</td>
<td>analytical, abstract, systematic, creative</td>
<td>Smalltalk</td>
<td></td>
</tr>
<tr>
<td>Mario</td>
<td>conceptual world</td>
<td>object-oriented, AI</td>
<td>use</td>
<td>analytical, abstract, can see multi-levels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ross</td>
<td>imagination</td>
<td>functional, AI</td>
<td>LISP vs other</td>
<td>abstract pattern matching, creative</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
It is striking that the respondents' mental models appear to correspond, with some overlap, to the programming paradigms that they prefer. Melissa, Claire and Isaac, who tend to model at more concrete levels, are all procedural programmers. Lorne, Kyle, Mario and Ross, at the abstract end of the scale, like languages like LISP, Scheme and PROLOG, used in the field of artificial intelligence. Those in the middle, overlapping the previous grouping -- Mario, Kyle, Lorne and James -- like the object-oriented languages Smalltalk and C++. Isaac too is beginning to shift toward that paradigm. This analysis of the respondents' mental models leads to a re-examination of the role of programming paradigms. Did the respondents choose the language paradigms they prefer because of innate mental models, or has the use of a particular paradigm shaped the ways in which they conceptualize the act of programming? As mentioned in the discussion of programming paradigm, most of the respondents feel that both principles are true. All of them feel that programming in a particular language or paradigm shapes one's way of thinking. However a wide breadth of experience allows programmers to choose languages that fit the way they think.

It appears that the respondents' preferred mental models shape many aspects of their thinking about programming and teaching programming. The models that the respondents use when teaching naturally tend to match the level at which they are modeling programming. Thus Melissa draws pictures of storage locations in order to help her students understand programming. Claire, when explaining a program to me, drew a module hierarchy diagram, which shows the relationships between modules in a program's structure. Isaac models programming by describing the detailed syntax and semantics of languages, and by helping students understand existing algorithms so they can build on them. James presents various mental models to his students: beginning with the most conceptual level, the function of a
program, and becoming more concrete as necessary to discuss specific aspects of program structure. Lorne’s programming model incorporates both pictures of physical world objects and program details. As well, he stressed the importance of thinking of statements in high-level languages as groups of machine-level code. Neither Kyle, Mario, nor Ross drew pictures when discussing program code. Neither did they discuss models used when teaching, but rather talked about ideas and problem domains.

The ways in which the respondents categorize languages appear to be strongly shaped by their mental models. Melissa categorized languages based on awkwardness. That is, she judges them based on the ease with which she can manipulate them. Her favourite language is assembly language because it allows her to get as close as possible to the physical machine. Claire categorizes languages in terms of their structure and syntax. She likes Pascal, a modular, highly structured language that allows her to teach the concepts of structured programming. Isaac categorizes languages in terms of paradigm. The amount of overlap he sees between paradigms and the fact that his description of the procedural paradigm is so highly differentiated points to his sense of programming as an integration of various complex detailed systems. Isaac’s favourite language is Modula-2 because it provides discipline in the complex programming environment. However, he sees great potential for C++, which supports both procedural and object-oriented programming, because, as he said, some problems fit a procedural solution while some fit an object-oriented one. James and Lorne categorize languages in terms of use and both like the object-oriented paradigm because it allows them to closely model the real world. James favourite language is C++ because it seems to be like the real world. One of Lorne’s favourite languages is Smalltalk because it allows him to think about an exact model of the world. Lorne mixes languages based on

What Teachers Know about Programming
the requirements of the problems he is trying to solve and appears to move with
great ease from one level to another. Kyle categorizes languages in terms of
paradigm. In particular, he likes Smalltalk and PROLOG, because with them he can
develop

\[ \text{a very high-level abstract symbolic representation of the world} \]
\[ \text{which has a relatively close mapping to the classical ways that we} \]
\[ \text{... model the world internally.} \]

Mario categorizes languages in terms of the ways in which he uses them to
implement solutions and teach various principles. To him, good languages map
the problem domain. He finds that Smalltalk, Scheme and C++ allow him to easily
implement solutions for the problem domains with which he deals. Like Lorne,
Mario seems to move easily from one level to another and said that \textit{the ability to}
\textit{move quickly from one level to another} is one of the attributes of a good
programmer. Ross sometimes categorizes language as LISP and other. LISP to him
is different from other languages because it allows him to implement ideas that are
[dreamed] \textit{up out of the air}; because it gives him a sense of \textit{inspired creativity}. He
said, however, that he is really \textit{wild about} a language that doesn't exist yet. That
other language \textit{doesn't have a lot of parentheses, has a very smart interface, doesn't}
\textit{require compiling} and doesn't get in Ross's way. It lets him build whatever he
wants \textit{out of the air}.

The respondents' ideas about the best first programming language reflect the mental
models they have of programming. Melissa suggests that students should learn
assembly language first so that they can better picture what is going on inside the
machine when coding in other languages. Claire favours Pascal because it is a good
introduction to \textit{structures} and \textit{modular programs}. Isaac also would teach Pascal first
because of the structure and limitations it imposes on the programming process.
Lorne would teach assembly language first. Although his mental model is abstract, he feels that an understanding of the way in which high-level languages can be broken down into lower level instructions is vital because high-level languages are just expansion on assembly language. Kyle feels that Smalltalk or an AI tool should be taught first in order to give students the ability to model the world and to show them a creative view of what can be done with a programming language. Mario ridiculed the heated discussion about best first language. He feels that decisions about language should be grounded in the principles which one is trying to teach.

The mental models that the respondents use when programming seem to shape their ideas about the attributes of good programmers. All of the respondents said that programmers require intelligence or the ability to think well. Upon close analysis, however, it appears that the meaning of the term intelligence and the reason for its importance differs from respondent to respondent. Melissa feels that programmers need to be able to think logically, and need a strong sense of cause and effect, reflecting her model of physical linear programs. Claire, for whom programming means coding and who concentrates on the level of program code, said that programmers must be able to think well, by which she means be able to translate ... the ideas into code. Isaac thinks that programmers need above average intelligence so that they can grasp and remember a vast quantity of facts and details. James, who is most concerned with modeling the world, maintains that programmers need to be able to think well independently, using their knowledge to model application areas. Lorne described programming as a thinking person's game. His description of the necessity for programmers' to have the ability to organize their thoughts into an overview of an entire system and break it down into all the component pieces mirrors his tendency to move from one level of model to another.
Kyle, Mario and Ross appear to have more abstract interpretations of the term intelligence than the other respondents. Kyle feels that programmers have strong analytical and deductive reasoning skills, and Mario's interpretation of the intellectual skills required by programmers is that they need to be able to think analytically and synthetically. Ross's sense is that good LISP programmers are **attuned to intellectual issues in some sort of abstract pattern matching kind of way.**

The two respondents who, like Ross, tend to use more abstract models when programming, are the only ones who echoed his feeling that programmers need to be able to think abstractly. Kyle said that programmers are good abstract thinkers and Mario feels that not only must programmers be able to create mental models, but they must also be able to see things at multiple levels of abstraction. As mentioned previously, however, only Ross differentiated between the skills required by those who program in one paradigm and the skills required by those who program in another.

The respondents who tend to use more concrete models mentioned many more personality traits required for programming than did the other respondents. Claire and Melissa agreed that programmers need to be patient. Melissa also feels that programmers need to be hard workers who are persistent and can concentrate well on details. Isaac feels that programmers need the kind of temperament that allows them to apply discipline to programming. Claire agrees that programmers need to be systematic and have methodology. Although Kyle, at the abstract end of the scale, echoed Claire's opinion that programmers need to be systematic and methodical, he was the only one of the other respondents to mention these traits.

Several of the respondents said that programmers need to be able to see the big picture as well as the details when programming. Lorne and Mario, who tend to use
various levels of mental model when programming, feel that this ability to move between levels is an important attribute of good programmers. James and Melissa also feel that programmers need to be able to solve problems at various levels. James draws a strong distinction between analysis and coding. He said that both activities are necessary, but feels that the former is by far the more important. Melissa, although she too recognizes the need for multi-level thinking, feels that it is more important that programmers are able to concentrate on details.

Respondents from Melissa, who models at the most concrete model level, to those who model conceptualizations of the world, mentioned the need for programmers to have good planning skills. Mario mentioned Alan Kay's analogy, relating programming to building a Gothic arch. Anybody can build a wall, Mario said. To go beyond that, though, -- to build a Gothic arch -- one must plan the whole structure ahead of time. Ross alone, who prefers to model the world of the imagination, feels that the kind of programming he does is not about planning, but about experimentation and creativity.

Isaac, Lorne and Kyle echoed Ross's sentiment that creativity is a part of programming. Isaac compared programming to painting a picture. Lorne compared it to composing a symphony. All four, however, volunteered the information that they themselves cannot draw or paint. It appears that to them the creativity involved in programming is different from the creativity involved in drawing or painting. As Ross said, he does not have the talent to sculpt or draw, but thinks that in programming he has perhaps found his artistic medium. When I asked them to evaluate whether or not people they know would make good programmers, three of the respondents used artists as examples of people who would not make good programmers.
Melissa is the only respondent who said that she herself is not creative. It is interesting to compare Melissa’s description of a person who would not make a good programmer with Ross’s description of how he feels when programming.

Melissa’s description of her friend, an artist:

*She can take a canvas and put a little blob here and a little blob there and after a while it starts to turn into something. And I’m not sure that she knows all along that that’s what she’s going to end up on .... She’s a hairdresser. So she cuts a bit here and cuts a bit there and looks at it and says, “Oh, maybe, OK, I’ll cut a little more here and I’ll change it here,” rather than sort of plotting the whole thing out.*

Ross’s description of programming:

*To caricature it a bit you get the barest inkling of an idea and you try it out right there and then on the machine and that action leads to a sense that it’s a dead end or an improvement or maybe you’re quite satisfied with it, but the process is very immediate. It’s have a thought, try it out, have a thought, try it out.*

Melissa, who when programming models the physical reality of the machine, appears to have a very different notion about programming than does Ross, who, when programming, models ideas that he dreams up in his imagination. To Melissa, planning and method are important, whereas to Ross, programming is creative and interactive *with a capital I*. This difference in thinking about programming is echoed in Melissa’s and Ross’s very different teaching styles. Whereas Melissa shapes and structures her students’ programs and uses physical models so that they understand what is happening in the machine, Ross [swims] *through this sea of alternatives* with his students discovering and inventing patterns that can be modeled on the computer.

The respondents mentioned three other attributes of good programmers. Lorne, Melissa and Claire said that people must, above all, be motivated to learn programming. Each of them mentioned people who, although they might have the
ability to learn to program, were not interested in doing so, and would therefore make poor programmers. Mario, Melissa and Claire feel that programmers need to have good communications skills. Mario and Melissa also mentioned that programmers need to have confidence that they will be able to solve a problem or find a bug.

It appears that the only quality felt to be important by all the respondents is a certain level of intelligence, which seems to mean different things to different respondents. Additionally, those at the abstract end of the scale feel that programmers need to be good abstract thinkers, whereas those at the more concrete end are concerned with personality traits such as patience and persistence. There is a feeling among some of the respondents that programmers need the ability to see problems at various levels of abstraction. As well, the ability to plan was felt by many to be an important attribute.

The respondents had various opinions about predicting programmer success. Claire feels that anyone can learn to program. Isaac, Mario and Melissa disagree. Mario said that whereas anyone could learn to use Hypercard or Toolbook, not everyone would be able to go beyond that. Both Lorne and Kyle think that while some people have natural programming abilities, with hard work and much training others could learn how to program as well. Ross is unsure whether programming ability is innate or programmed. Most of the respondents feel that they can predict in a matter of minutes whether a person will be a good programmer and most, when asked to tell me whether people they know would make good programmers, were able to do so easily. Isaac, concerned with the complex interaction of programming systems details, said that he feels that he can predict which students will be good programmers by listening to their speech. Those who do not pay attention to grammar and syntax, he feels, will not be good programmers. Both Lorne and
Mario, who tend to model real world problems when programming, evaluate people on the basis of their problem solving skills. Both feel that people who are willing to try a problem-solving approach, whether ... right or wrong, as long as they [try] to solve the problem, have a good chance of becoming successful programmers. When listening to people talk about problem solving, Lorne evaluates whether they are able to break a problem down into sub-problems. Mario said that he is looking to see whether people are able to set goals for themselves and evaluate their success at reaching those goals.

Interestingly, when talking about programmers and their traits, six of the respondents mentioned that they separate programmers into two distinct types. Although each respondent labeled the groups differently, it appears that they may be talking about the same groups of people. Those in one group are goal-oriented and, as Ross said, keep the engines of society oiled and working as they should. Claire calls them applications programmers; James calls them business programmers. For Ross they are the programmers who work in the block structured world. Isaac calls them the neat because they tend to wear business suits and work in a corporate environment. The programmers in the other group, the ones that Mario says want to push things to the limits and explore the boundaries of the system, seem to be more interested in programming for the fun of it; for the opportunity to learn and explore that it affords them. These programmers Claire calls systems programmers and James calls scientific programmers. For Ross they are the programmers who inhabit LISP-world. Melissa calls them nerds. Although the respondents have slightly different definitions of the two types of programmers, they seem to be talking about the same differentiation. As Mario said, tension exists between the desire to explore and enhance programming tools and the need to focus on a goal. It
requires exceptional ability, he said, to be able to marry the two aspects of programmers' personalities in order to get the task at hand done.

Thus it appears that the mental models the respondents use when programming strongly shape many aspects of their thinking about programming and teaching programming. In particular, the models they use when teaching, their views about language and their estimation of programmers' attributes appear to be shaped by the mental models they tend to use.

Conclusions

This study set out to investigate the answers to the following research questions:

How do teachers organize their knowledge about programming?

How do teachers organize their knowledge about teaching programming?

How do teachers' constructions about programming integrate with their constructions about teaching programming?

The following conclusions may be drawn from the analysis of discussions with eight programming teachers.

1. The respondents' knowledge about programming and their knowledge about teaching programming appear to be constructed within the same conceptual frameworks. The two knowledge domains seem so inextricably intertwined that the relevant research question seems to be: Within what frameworks do teachers construct their knowledge about programming and the teaching of programming?

2. The respondents appear to construct their knowledge about programming and the teaching of programming within conceptual frameworks, five of which are:
1. life experience,

2. social context,

3. desire for control,

4. paradigm experience, and

5. mental programming model.

The conceptual frameworks which emerged from this study appear to be interlinked. Particular frameworks appear to have more importance in shaping some respondents' knowledge than others.

3. The respondents appear to use mental models of the programming process. The models used by the respondents range from the concrete to the abstract. Particular respondents tend to use particular models. Most also at times employ models at varying levels of abstraction.

4. The mental models that the respondents use appear to relate to the paradigms in which they prefer to program. It is unclear whether the respondents choose their paradigms based on an existing mental model or choose a model based on the paradigm in which they work.

5. The respondents feel strongly that it is very difficult for programmers to shift from one paradigm to another. They feel that the paradigms are very different from one another and that programming in a particular paradigm shapes the way a programmer thinks.
6. Because of the difficulty of shifting between paradigms and languages, the respondents feel that the decision about which language students learn first is critical.

Limitations

The study's conclusions are based on the respondents' articulated perceptions only. No triangulation was done to see if the teachers' perceptions of what happens in their classrooms fits others' perceptions.

Many of the respondents teach at the same school, know one another and know the same students. Thus similarities among their views may be a function of shared local knowledge. Interviews with teachers at different schools may uncover different perspectives.

Because the study was exploratory in nature, the interviews had a very broad focus. There are therefore many areas which it would be useful to investigate in more detail.

Implications

The results of the current study have implications for several areas: further research, the evaluation of existing literature and teaching practices.

As this study progressed, the research questions changed shape. It appears now, because of the interlinking of the respondents' knowledge about programming and their knowledge about teaching programming, that a better research question to ask would be *Within what conceptual frameworks do teachers construct their knowledge about programming and teaching programming?* A study designed to
investigate this question might be more apt to uncover frameworks which have shaped teachers' knowledge.

Furthermore, each of the frameworks uncovered in this study appears to offer a rich area of additional research. Studies investigating the ways in which each of the frameworks (life experience, social context, desire for control, paradigm experience and mental programming model) relate to teachers' knowledge would likely lead to deeper understandings. Caution must be taken, however, not to concentrate on one area to the exclusion of others. The purpose behind focusing on one framework would be to increase depth of understanding so that the overall picture can be better informed.

Researchers often do not make explicit their views about programming or the perspective from which they are approaching issues such as prediction, teaching methods, and choice of language. It appears that there exist well-developed frameworks within which the respondents construct their knowledge about programming and teaching programming. It may be informative to evaluate existing research studies with attention to the frameworks and perspectives within which the researchers are conducting and presenting them. In particular, the emerging literature regarding the paradigm shift and the need for an alteration of teaching practice based on a change in paradigm should be judged in light of the differing perspectives that researchers schooled in non-procedural paradigms are likely to have.

The respondents have strong opinions about the differences involved in programming in different paradigms and about the difficulty of switching from one paradigm to another. It may be valuable to examine the current practice of teaching the procedural paradigm extensively, and the arguments against this practice, in
light of these strong opinions and with an understanding of the purpose for which
programming is being taught. Furthermore, if, as the respondents feel, the choice of
language is critical, the standard practice of teaching procedural paradigm initially
must be re-examined.

The respondents have strong mental models of programming that are reflected in
their teaching practices. Students too use mental programming models. An
investigation of the differences and similarities between teachers’ and students’
models may lead to a better understanding of the ways in which teaching practices
can be altered in order to be of most benefit to students.

Four additional issues arising from this study appear to be worthy of further
investigation:

1) The relationship between mental model and paradigm.

Some research has been done regarding programmers’ use of mental models
(Pennington, 1985). It appears that a relationship exists between programmers’
mental models and the paradigms in which they work. Further investigation of
this area may prove fruitful.

2) First language.

Several schools now teach a non-procedural language as students’ first language
(Abelson and Sussman, 1985; Reid, 1992). A study of these students’ programming
success compared to those learning procedural programming may expand our
understanding of the influence of students’ first language. In particular, an
examination of students making the shift from one paradigm to another may prove
enlightening.

What Teachers Know about Programming
3) Programmer attributes.

The respondents had mixed opinions about the attributes required by good programmers. A study of the attributes of expert programmers in various paradigms may help differentiate the area. Furthermore, an examination of predictor variables related to students' success in various paradigms may uncover useful information about the ways in which success can be predicted.

4) The two types of programmer.

Almost all of the respondents differentiated between two distinct groups of professional programmers. Further investigation of the qualities of each of these groups may lead to a better understanding of the attributes that programmers require.
REFERENCES


Clarke, Anthony and Donn, Stuart. *Novice programmers perceptions of two instructional methods; the first teacher centred, the second student centred*. Unpublished manuscript.


Qualitative methodology: Firsthand involvement with the social world. Chicago: Markham Publishing Company.


What Teachers Know About Programming 183


*What Teachers Know About Programming* 184


March, Richard. (1989). *Coping with the PC slowdown*. PC Week, 6(38), 119-120.


Soloway and James C. Spohrer (Eds.), *Studying the novice programmer*. Hillsdale, New Jersey: Lawrence Erlbaum Associates, Inc.

Ralston, Anthony and Shaw, Mary. (1980). *Curriculum '78 -- is computer science really that unmathematical?* Communications of the ACM, 23(2), 67-70.


*What Teachers Know About Programming*


APPENDIX A
GLOSSARY

Algorithm: A finite set of step-by-step instructions for solving a problem

Application: The job or task that the user wants the computer to perform

Arithmetic Logic Unit: The part of the computer that performs the mathematical and logical functions

Array: A structured data type

Artificial Intelligence: The field of computing that investigates the use of computers to simulate human reasoning

Bit: A binary digit

Bug: An error in a computer program

Byte: Eight bits

Coding: The process of writing the computer program instructions in a particular programming language

Compiled language: A language in which the program code is translated to machine code all at once (See interpreted language.)

Computer Assisted Software Engineering: The process of creating software with the assistance of specially designed tools

Computer program: A series of instructions that tells the computer what to do

Debugging: The process of finding and correcting a programming error

Desk Checking: Tracing the steps of a computer program by hand on paper

Environment: The machine upon which a programming language is used and the way in which it is implemented

Expert Systems: The field of artificial intelligence that deals with using computers to simulate the knowledge of an expert

Flowchart: A graphic tool used to design and document the steps of a program

FOCUS: A high-level programming tool
Fourth Generation Language: A language which does not require the user to have knowledge of low-level detailed program instructions

Hardware: The physical components of a computer

Hexadecimal: Base 16.

High-level language: A programming language that uses instructions that closely resemble human language and mathematical notation

Inference Engine: The facility built into a language used for artificial intelligence programming which performs logical operations

Interpreted language: A language in which the program code is translated to machine code one line at a time (See compiled language.)

Linked List: An abstract data type

Loop: A programming construct which causes instructions to be performed repetitively

Mainframe: A large scale computer

Network: To join several computers together so that they can share common data and programs

Oracle: A database package.

OS/2: A computer operating system

Pixel: The smallest part of a display screen that can be individually controlled

Pointer: A data type used as an internal address reference

Programmer: The person who writes the instructions, or code, that tell the computer what to do

Programming language: A set of written symbols that instruct the computer to perform specific tasks

Queue: An abstract data type

Register: A part of the central processing unit which stores information for immediate use

REXX: An IBM mainframe system language

Software: The instructions that direct the operations of a computer
SPSS: A statistical package.

Stack: An abstract data type

Support: A job which involves helping users solve computer problems

Systems Programming: Writing programs that control the computer's operating system

Third Generation Language: A high-level language (e.g. Pascal)

UNIX: A computer operating system

Windows: A graphical user interface
APPENDIX B
SAMPLE INTERVIEW TRANSCRIPT

Kyle: Mar 17/92

D: Do you know what I'm doing?

K: Sort of. My understanding is is that you're working on a Master's Thesis in Education and you're investigating something about Computer Science teaching.

D: mm-hmm

K: And pedagogy.

D: Yes. Yeah. So I'm looking at um teachers', people who teach programming's, view of programming and their view of teaching and what makes a good programmer and that kind of thing.

K: Mm-hmm

D: And the sense of the interview is not one where I sit and ask you questions and you answer this form of questions that I have here.

K: Right.

D: But it's more what's called a 'rapport' interview. So we know each other and we're not going to pretend that we don't know each other, so any lingo you want to use or referring to things that I know about, like "I teach 444" or whatever it's called, you can just do that and you don't have to worry about the fact that there's somebody else listening. Ok?

K: OK

D: All right. How did you come to be teaching in Computer Systems?

K: Day time or night time?

D: Either. Both, actually.

K: I started off uh ... The Continuing Ed teaching started because there was a missing Pascal teacher many years ago.

D: Oh.
K: in 1985 or 1986 and uh they knew, uh A and B knew that I knew Pascal. I think B passed my name on to A and A called me up and asked me if I'd like to teach the Pascal course and um so yeah, that's the way it started.

D: And had you been a student at BCIT already?

K: Yes. Before that I had ... My educational background is that I did the second year of BCIT's CST program, graduating in Expert Systems option and as you know I recently wrapped up my BSc in Computer Science at SFU.

D: And then how did you come to be teaching day school?

K: Well, in the night school I eventually moved from Pascal to focus on what are my specialties, which is AI programming techniques: LISP, Prolog, Knowledge System techniques in general and as an adjunct, Smalltalk and object-oriented programming which are skills I've been playing with for years. And so I was um uh sort of had a long history and experience in Smalltalk and OOPS programming and when C wanted to start up um a Smalltalk class in the daytime, he asked me if I'd like to teach it.

D: Mm-hmm. So when did you start programming in Smalltalk?

K: Um... maybe I'm guessing, but maybe 1987, 1986, but my experience with object-oriented programming goes back to I guess 85.

D: And that would have been in ...

K: In the expert systems option.

D: OK

K: Cuz these concepts, basically or have been largely influenced by a lot of AI programming work.

D: Mm-hmm mm-hmm So would you make a distinction there at all?

K: Um, well a lot of the concepts that you find in in object-oriented programming are also found in the concepts of frames and frame-based programming. Um yeah there are some technical distinctions, but they're minor and the uh similarities are a lot larger. Yeah. Like a lot of the, a lot of the stuff that you see in object-oriented programming ... There's different levels of it. One of the levels that you can look at object-oriented programming is as a a way to represent the world, you know, and knowledge about the world and things in the world, and um in in AI work, and in Expert Systems work, you're looking at the same problem, you're looking at how do we represent knowledge and what are different taxonomies or ontologies, ways to represent things and that's uh uh the
similarities between frame representations and object representations are very very similar.

D: Mm-hmm, mm-hmm. Have you taught ... So you've taught Pascal, Smalltalk, ...

K: LISP

D: LISP

K: Prolog and uh uh Knowledge Systems courses, in building knowledge systems.

D: OK. Anything else that you've taught either here or elsewhere?

K: Uh, yeah. APL, here, I taught APL. Uh, to the daytime CST students ...

D: Oh right.

K: some years ago.

D: Oh yeah, I remember that.

K: Yeah

D: I'm just going to write some of these languages down ...

K: Sure

D: cuz I'm going to ask you about, something about them in a bit.

K: Um. and I have done um Smalltalk courses for outside groups, people like BC Tel.

D: Mm-hmm. Mm-hmm. How did you first learn how to program?

K: At university.

D: Mm-hmm.

K: Yeah. I um chose Computer Science because it seemed to be a um good field in terms of employment. I guess I made this decision like in 1978.

D: OK, so you were just coming out of high school?

K: No, I was coming back from India.

D: OK.
K: Yeah (laughs)

D: But this would be so you you after high school ...

K: Oh, I didn’t well, I quit high school and I spent several years living on the road um travelling North America and whatnot and had many adventures, eventually ended up in India and um it was when I came back from there in 78 uh that I started university.

D: You chose Computer Science having come from India?

K: Yeah

D: Because it seemed like it offered good employment?

K: Well, I’ve been meditating for years, since I was a teenager and when I was in India, my teacher recommended that I study something in which I could make a lot of money easily (laughs).

D: That sounds so so the opposite of what I would think a teacher in India would say.

K: Yeah, cuz I mean the basic principle in in meditative traditions or in yoga traditions is that you want some free time in order to be able to devote to spiritual practice like meditation, to service, and things like that and if you’re so poor that you have to spend all of your time working, then you won’t have any time free to be able to devote to service or meditation etc. and uh so it’s actually a very common teaching to uh work efficiently, to learn something that you can make money easily with, so that you have free time.

D: Mm-hmm mm-hmm. Has that worked out? Do you have free time?

K: Yeah.

D: Yeah? That’s not my sense of computer programmers at all. (he laughs) People with lots of free time, no.

K: Yeah.

D: Hmm. So ...

K: So, I uh

D: you went to SFU?

K: I chose Computer Science because it looked like um a) it looked like it was a useful field to get a job easily in and although I knew nothing about what
the work would be like, um it it interested my vaguely, I had an aptitude for scientific and mathematical things.

D: OK so, you knew that you had an aptitude for Science and Math from ...

K: Yeah

D: high school and that kind of stuff?

K: Yeah and just from my interests.

D: Yeah

K: I always loved Science. My first interests, you know, ever since I was a little kid.

D: Mm-hmm. And so it was your sense that Computer Science was going to ...

K: I chose ...

D: would require the same aptitude?

K: I chose it primarily because um I chose it primarily because it seemed to me I would be able to get a job in the field and that it paid well.

D: Mm-hmm

K: And um and secondly because it seemed in some way to be sort of scientific or technical and that seemed to fit with my interests.

D: Yeah. OK. So, you did that for a number of years..

K: Do you want a basic history?

D: Yeah, I guess so,

K: I, uh ...

D: cuz I want to ask about your sort of work background as well.

K: Sure. I did uh 3, 3 ... I did maybe 4 semesters in a row at SFU, like a year and a third of Computer Science courses and so that's where I was first exposed to programming.

D: OK. What was your first language?

K: PL/I
D: Pl/I?
K: Yeah.
D: Oh, interesting.
K: You know and it it in those days and still today a relatively good first language because it was um uh you know well-structured, um rich, etc.
D: Yeah, yeah. So that's ...
K: That ..
D: Go ahead.
K: That was the first language of choice that they chose to teach students at SFU in those days.
D: And so you would say that that's a reasonable choice for a first language?
K: Yeah, at the time it was a good choice, cuz you know, it's structured programming constructs and stuff like that.
D: Yeah Yeah. It... Does that imply that it wouldn't be a good first choice now?
K: Uh, it's a little bit top heavy in that it's got a lot of features? And well, it's hard to say. Like on the one hand it's got a tremendous amount of features. That could make it potentially overwhelming for a student, but then again you can look at PL/I in terms of a subset language and chop out a lot of these other features and then as you want to introduce new features to students for example pointer manipulation or recursion the language supports that as well so in that sense the students could move from simple programming to more complex uh programming techniques all within the same language so there's that consistency.
D: Mm-hmm
K: So, you know, it has advantages that way.
D: Yeah yeah
K: And as well of course there's a lot of hooks to different types of file processing um which was useful and practical.
D: Mm-hmm
K: Yeah. So after uh 4 semesters, I joined the co-op program at SFU and went to work at ICBC and I chose ICBC because a returning co-op student, who I asked about, you know, places to work uh said that they had a really neat text editor there. It was like a full-screen editor in which you could move a cursor in two dimensions and stuff like that.

D: Wow (we laugh)

K: And I went wow!

D: I remember those days.

K: Because up until that point we just used an MTS line editor..

D: Yeah yeah

K: for for editing. So uh so that's why I chose ICBC. And my first semester at ICBC as a co-op student was as a PL/I programmer and it was a great experience because um I was introduced to large formal methodologies, a shop that had to develop really big systems that um you know used IMS and um formal database policies, formal uh design and analysis policies and um people worked in teams and uh so it was an excellent, really an excellent experience.

D: mm-hmm mm-hmm

K: Um, came back after that semester, did another semester or another semester or two I can't remember, of Computer Science.

D: Yes.

K: Then I went back to ICBC again for another co-op term and this time I asked to work as an APL programmer because by this time I'd learned APL and um uh it was um a much more interesting intriguing language to me than conventional 3GLs.

D: In what way?

K: uh its expressiveness. The ability to do very complex things in a very compact amount of statements. Um its consistency to the um to the paradigm of um you know matrices, scalars, matrix uh manipulation. Um that's one of the things that intrigued me about it, there were like it it definitely like was in a particular paradigm and that is, of manipulating matrices.

D: Mm-hmm.

K: And uh that was interesting to work with.
D: mm-hmm, mm-hmm. Anything else about why it was so intriguing?

K: Uh, well, I think I'd just be repeating myself and that would be that you could you could do things quickly in it.

D: OK

K: In a powerful language.

D: OK Do you still feel like that about APL?

K: Yes. Mathematical type problems. Like at the time, spreadsheets didn’t exist. This was in the 1970s. And um so in uh at ICBC it was used for a lot of financial analysis work, you know, things today you would use a spreadsheet for, but at that time it was uh many times faster than using any other tool except some of the report writers that were available uh for stuff and it was even faster in those days for building transaction applications than um most other tools available. Um IBM APL included a hook to a um like a screen interface uh and also hooks to IMS so that we could use it for actually making, you know, a transaction application which would display out and read records to IMS databases. So we used IPL uh APL for doing uh prototypes of um applications which would, which actually went into production like at selected sites. An example would be um um when you wanted to make an appointment with an adjuster there was a system that would automatically do like adjuster appointment management.

D: Mm-hmm Mm-hmm Wow. That’s amazing I never would have thought that - APL.

K: Yeah. And and our group that ... it was an APL group, you know, it was like like the leading edge group in the Systems uh department because we were using report writers in APL and just spitting out applications ten times faster than anybody else.

D: Neat

K: Yeah

D: So ...

K: And after that um uh what I really wanted to do was go back to India um all along my goal had been to save enough money to go back to India for say 5 or 10 years and stay there for a good long time so after my first that second work semester at ICBC, I elected not to go back to SFU, but just to stay on at ICBC and keep working, saving some money. So I did that for a while. Saved some money, went back to India for quite a long time. That takes me up to the, I guess, early 80s um I came back from India after
several years. I got sick and um had various adventures. Um eventually I uh did a business in which I would buy up review copy books from well, I couldn’t work regularly when I came back because I was sick for about a year afterwards with intestinal problems, the usual story, but I had a real bad long case of it and so I had to do some type of work um uh that I could do like at my own leisure. So I got into an entrepreneurial business in which I would buy up review copy books from university professors and sell them to a book wholesaler. Um and one day I was visiting BCIT doing that and I saw a poster about the possibility of entering second year directly for um certain qualifying students. And I realized right away I’d probably have those uh capabilities and uh I just dropped into the CST department to look into it and bang, that was my first introduction to the term Expert Systems and it immediately caught my interest and thus I started in the Expert Systems program. And I went through Expert Systems, graduated, got a job building Expert Systems in Prolog for a small startup company, worked for them for just a couple of months and then they ran out of money and went bankrupt. Uh then um I was hired to come here at BCIT as a Systems Analyst and uh while here my work has been very varied in terms of its uh what I’ve done over the years. Uh um a lot of work has been done in FOCUS, 4GL, um SPSS, REXX, um more stuff than I can remember, and um, uh, these days I’m using more a very high level tool like an EIS you know building tool ???? building tool in Windows and now we’re getting into Oracle because we’re doing the Student Info System in Oracle.

D: Right.

K: And I do consulting work on the side. Usually in Prolog and Smalltalk.

D: Oh yeah.

K: In fact always in Prolog and Smalltalk.

D: Yes. Um so what would be the last Prolog or Smalltalk program ...

K: Uh, the Prolog project that I’m working on at the moment is uh a knowledge system related to genetics. Um it’s being done with a consultant downtown who I’ve been associated with for quite a few years - Dr. L.

D: Oh yeah.

K: And uh the genetics department at UBC, um there is a uh genetic material, chromosome, can have abnormalities and geneticists and there are a is it 26 different chromosomes in a human being. And geneticists look at these chromosomes and they’re trained to be able to identify what the abnormalities are? And then they record these abnormalities using a notational system called ISCN.

What Teachers Know About Programming 202
D: I think he came to a Professional Day ...

K: Dr. L. did?

D: Dr L. did.

K: Kay, so you know the story anyway.

D: Yeah.

K: I'm working on that project.

D: Neat.

K: So it's a very interesting project and um uh it's going to be extended for leukemia abnormalities now. There's two types of abnormalities: congenital and acquired. Acquired abnormalities are those you get through disease, like leukemia and it's a chicken/egg scenario. I'm not a geneticist like I don't know whether you get the ...

D: Right

K: you know, what comes first.

D: Yes

K: But anyways when leukemia is present, there are genetic abnormalities present as well.

D: Mm-hmm

K: So I'm working on a project related to that.

D: Hmm. Wow wow. And that's Prolog?

K: Yeah

D: Um, so you've been programming for a fairly long time. You're programming now.

K: Since 78.

D: Mm-hmm. Kay. What do you like about programming?

K: Creativity. For me it's a um for me it is a an outlet for uh creative expression for um uh well for constructing something and uh I like the I also like the fact that you're solving a real world problem. I'm also a musician and um.
D: Um so you were saying that you’ve been a musician since you were a young teenager.

K: Yes and um although composing music doesn’t have the same sort of practical value as uh writing programs does for me, I still enjoy the creative expression process, but definitely part of what interests me in programming and Computer Science, is that it has, it has something to do with the world, you know manipulating the world or solving a problem. And um my real interest these days, my particular research interest is in what are for me socially relevant applications of knowledge systems and AI. I did a research project last year for Mac Blo, research in which I developed a knowledge system for treating pulp mill effluent. And of course there’s the genetics problem and I’m particularly interested in these types of applications of the computer. Um now, what else do I enjoy about computing, or what attracts me to it? Uh, part of it is um, I guess an abstract esthetic quality similar to um in mathematics, if you like mathematics, and if you’ve ever seen a beautiful theorem or a proof or something, there can be a sense of pleasure in uh the esthetic beauty of that. And it’s similar in Computing Science for me. And in fact there’s a correlation in my mind and in my emotions between the sense of pleasure I get and the esthetic qualities of the language. And the esthetic qualities of the language to my mind are influenced by the um uh by how closely the language is build on some type of theoretical basis or is consistent to some type of theoretical basis. For example, SQL is a very simple language and the relation and model are very simple, but they’re built on a very simple, you know, framework of like the predicate calculus. And it’s consistent, it is ... and there’s a beauty in it to me for that. Uh Object-oriented languages, Smalltalk in particularly, is completely consistent to the object paradigm, which is built on a sort of a consistent view of the world of class hierarchy and inheritance etc. and so there’s a ... I find an esthetic beauty in that. Prolog, consistent to the logic programming paradigm, same thing. And other languages which are more murky, be it and just are built to get the job done, be they PL/I or FOCUS or LISP or whatever, I have a, like a sort of degrading level of esthetic beauty associated with them in my, in how I respond to them.

D: Yes. What do you dislike about programming?

K: Um I dislike programming languages which make it difficult to achieve some functional goal. For example, if I wanted to write a GUI and the language either had the facility, but it was very awkward to use, or didn’t have it, that would make me dislike it. That’s sort of a very feature-level thing though so ...

What Teachers Know About Programming 204
D: Yeah

K: In a general way, um programming requires um mental activity of a certain nature, intellectual activity and I'm aware that mind or consciousness can operate in different modes. It's possible to be in a state of awareness or consciousness in which there's, you're aware of the present moment, you're aware of your bodily sensations and emotions etc., you can respond and talk, but there's not a lot of intellectual mental activity going on. Programming and intellectual thought require a movement of the mind in a particular way. And if I do that too much - thinking - um I'll get mentally fatigued and um won't experience the same level of bliss and happiness which I can experience say through um being having less mental activity. To put it another way, if I meditate a lot and experience mental stillness and quiet regularly, a sense of peace and joy will arise in my spontaneously and I'll feel a sense of integration and wholeness.

D: mm-hmm

K: And um the more I engage in continuous mental activity, say from the moment I wake up til the moment I go to bed every night, the more I feel I'm out of touch with feeling joy, peace, or a sense of integration. So what I dislike about programming is uh that it can um it can create too much mental activity in and not feeling completely integrated emotionally, mentally, spiritually.

D: So those components are missing.

K: It doesn't have to, I mean like it's a matter of, a matter of balance ...

D: Ok, yeah

K: But, but but programming and mental activity in general is something if you do all the time, I would not be in the optimal spiritual emotional state that I'm capable of being in.

D: Right

K: Because to be there I also need to do things like physical exercise and spiritual exercise in order to be peaceful and healthy.

D: Mm-hmm. Anything else that you don't like?

K: Uh, as I become more skilled over the years, uh I dislike doing work which is trivally easy for me.

D: Yes
K: It's very difficult for me to find any programming problem now which requires any effort on my part intellectually. Um the only area in which I find that now is Artificial Intelligence, which, in my opinion, contains the most difficult and challenging and interesting problems in programming, Computer Science and um so this is nothing to do with programming, but about the nature of, you know, programming work, and that is that uh there's less challenge as the years pass in a lot of the work.

D: yeah. Hmm. OK. I've written down some of the languages that you have mentioned. I don't know if I've got all of them.

K: If you were to list a language,

D: Yes

K: I probably know it.

D: OK. OK. These are the ones that you've mentioned. Now I didn't put down the 4GLs and stuff. I don't know. Maybe you want to. So I'll put FOCUS and we can think about whether we want to put other things on here. What I would like ...

K: Uh you know all like it would almost be better by exception to list a language that I don't know. Snobol, COBOL, I know a lot of them.

D: OK, well, let's put down duh duh duh duh we've got APL, Smalltalk, LISP ...

K: And I'm aware from, I'm aware from knowing a wide variety of languages that there are major paradigms in how to approach programming problems. And I'm familiar with those paradigms and I'm able to compare them in my mind. Like I'm familiar with the difference between an object-oriented paradigm and a logic based paradigm or a procedural paradigm.

D: OK. Maybe rather, so what I was going to ask you to do was classify these languages.

K: Mm-hmm.

D: If I were to ask you to do that would you classify them paradigmatically.

K: Yes.

D: OK

K: That would be ... that's one way of looking at them. There's a few other ways you could look at them as well.
D: OK. Let’s Ok. So paradigmatically, what other ways might you be tempted to classify, if say you had to just pile ...

K: Oh, um, um 4, a lot of 4GLs and DBMS tools um sort of have a procedural flavour, but these days, but they also um they increasingly have a, like a, um, how would I call it, almost an active data dictionary approach to programming where maybe a data modelling paradigm? Like where for example if you’re developing some type of a relational uh database with transaction processing applications, it would be as simple as defining the relational model, the tables, and then say attaching, you know, like uh validation routines to the different um fields in your database, all at like an active data dictionary level and then you just ask it to generate screens that um, you know, do transaction processing for you and that everything else is built in. That’s, you know, only vaguely procedural. It’s almost as that’s a paradigm in itself. It’s kind of. .. It has a sort of a flavour of the object-oriented paradigm, but not quite.

D: Kay

K: Call it a data modelling paradigm.

D: Yeah I’ve actually seen ...

K: Or even ??? engineering paradigm.

D: Do I have that text? No. There are actually texts that separate it that way.

K: Yeah

D: That put it in a data ...

K: and that um is is like the de facto standard paradigm for a lot of like business development tools these days and rightly so because it’s an extremely effective and quick way to develop simple business applications for transaction processing and supporting. So there’s maybe that as a paradigm as well.

D: OK, yeah. So let’s get back to this paradigm.

K: Yeah

D: So you’re saying there’s um object-oriented, procedural, logic,

K: Logic, logic programming

D: Yeah. Um maybe this data driven model

K: Yeah

What Teachers Know About Programming
D: Can you define those first 3 ... in five words or less, no.

K: Ha ha ha ha Uh the logic programming paradigm is based upon uh trying to represent things in something like first order logic, the predicate calculus

D: Yes

K: Um with extensions um with like in the case of Prolog with particular extensions, almost procedural extensions which you need to understand such as backtracking, such as um that the order of clauses is chosen according to the order that they're typed in.

D: Yeah yeah

K: Um things of that nature.

D: OK So you say it's basically logic but there is there are extensions.

K: Basically the predicate calculus with certain extensions you need to understand like backtracking.

D: OK

K: Um

D: Are there other examples of logic programming besides Prolog? Of logic programming languages?

K: There are other logic programming languages, but I haven't worked in them and they don't come anywhere close to Prolog in terms of popularity.

D: OK

K: Um, uh the procedural paradigm is one in which you basically um are specifying a series of actions, um and that's how you would ... to me a procedural language is one in which you view that you've got um control structure, uh functions and procedures and uh you and a program is made up of a series of statements which are executed in sequence in which the actions are performed in the series in which the functions and procedures are called.

D: Mm-hmm

K: Um the object-oriented paradigm is one in which you view your data and the procedures as bound together into single entities called objects and uh in which there are qualities such as um all objects are organized into a hierarchy of class, class definitions which have the properties of inheritance for example.
D: Mm-hmm

K: And it's a it's a um it's a paradigm which reflects a very common which reflects um a common way that people think about a lot of the world.

D: Yeah

K: If you were to look into ways people think about the world in terms of generalization, specialization or whole/part categories, these are things which are fairly naturally reflected in the object-oriented model.

D: Mm-hmm

K: And um and besides that, in the object-oriented model you view programs as involving the passing of messages.

D: Yes

K: The sending of messages.

D: How do you view things like C++ or object Pascal or ...

K: As object-oriented languages.

D: Yeah?

K: Yeah. I mean the fact that you kind of drop out of the object-oriented model and go into the procedural to me just means that, you know, you could define that as a hybrid language in which you can work in the object-oriented paradigm or the procedural paradigm. Yeah. There are some other paradigms, you know, I must admit, like there's the pure functional paradigm, functional programming. LISP is not a functional language, but you could program in LISP as though it were a functional language. In other words you can limit the programming to a subset of pure functional programming.

D: OK. What makes it not purely functional?

K: There is constructs in LISP, um, that fall outside of the functional programming paradigm and by this I mean it like in the mathematical sense. Um in old LISP there were like were control constructs that fell outside of the um uh functional paradigm.

D: Like ...

K: Uh

*D: Like loops and stuff?
K: like the go like the go statement,

D: Oh, OK

K: There were loops. Um what are some other examples? Macros don’t... LISP, Common LISP macros don’t really belong to um uh to the uh functional paradigm. That’s another example.

D: So ... I’m hearing you when you say Prolog, when you talk about Prolog and you said it was um logic with extensions that are procedural, is that the same sense with LISP, it has functions, but ...

K: No, no, Prolog’s extensions aren’t procedural. Prolog’s extensions are are things like the fact that it backtracks.

D: Yeah

K: Um it I don’t know how to characterize that, like into a into some paradigm.

D: OK

K: Um

D: Sorry, I guess I’m, somebody else said something like that to me and

K: Yeah

D: I’m putting words in your mouth.

K: It has, I mean there, you know, besides thinking of it as just representing the predicate calculus, it has other features as a programming language

D: OK

K: But I wouldn’t necessarily call them procedural features, it’s just ...

D: other features

K: Other features (we laugh)

D: Yeah. Kay. Other. Miscellaneous

K: Yeah um now another way of programming which I’m pretty familiar with and I’m not really sure how to classify it is rule-based programming. Um which has uh some analogies to kind of like logic based programming, but it’s not quite the same thing. Cuz when I think of logic programming, I’m
thinking definitely of something that's quite related to the predicate calculus.

D: yeah

K: But when I think of rule-based programming, I'm thinking about being able to define um discrete independent rules or constraints um which are then processed through some type of an inference engine.

D: Mm-hmm

K: Be it backward, forward or a constraint based engine.

D: Yeah

K: And in rule based programming similar to logic programming I'm focusing on um like the specification of relationships or the specification of constraints as opposed to any as opposed to any procedural uh definitions.

D: Mm-hmm mm-hmm

K: So perhaps rule-based programming is is another paradigm.

D: Kay

K: And of course I have experience in it um in Prolog, cuz you can do like rule based programming in Prolog and also in Expert System shells which I've worked in.

D: Mm-hmm Mm-hmm Do you have a favourite language?

K: Yeah, it uh probably Prolog.

D: Mm-hmm

K: Maybe Smalltalk. It's a toss-up between those two. They're very different languages and you can do you know some things you can do very easily in one and vice versa.

D: Kay. What makes them your favourites?

K: What makes Prolog my favourite for and, and they're favourites for different reasons.

D: Kay

K: Smalltalk is my favourite in being able to do graphical user interfaces and in probably for like sort of general purpose programming problems. It's
extremely flexible and versatile and it doesn't get in the way. Some languages for difficult problems have a quality where they don’t scale up very well? For more difficult problems?

D: What do you mean? Scale up?

K: Um, they may appear to solve the problem adequately without too much programming effort for a small case or a small problem. But when you try to scale up the problem or extend the specifications, the functional specifications, it starts to be very unwieldy [sic] to develop the code and do the design.

D: Mm-hmm. Um can you give me an example of a language where that would be the case?

K: It's nothing, it's nothing inherent about the language, it depends on the particular problem.

D: OK

K: Um but I could think of languages, classical languages like C or Pascal, where if I wanted to do a sophisticated knowledge system, s... knowledge system, uh it might not work for a knowledge system which had a lot of bells and whistles in it.

D: yeah

K: You might take, be able to take one little specific functional spec in that and be able to do it, but when you start to having to create a bigger system, it starts to become very unwieldy in design and in code.

D: Mm-hmm

K: But Smalltalk, I've had the experience that it, it scales up to bigger problems easily, you know, you don't feel like you're fighting with it, that it's getting in your way.

D: Mm-hmm

K: That in some way that way of representing things in the object-oriented way, it it continues to extend and support you as you're going into bigger, bigger problems.

D: Mm-hmm

K: Um and I like Smalltalk because it's got a tremendous amount of built in code.
D: Yes

K: Um and the code is organized in a logical way in the class hierarchy. Um it seems to me that one of the things that we really need in programming is to be able to reuse code and to be able to buy like off the shop shelf software components. Um there’s no need in writing a piece of code that somebody’s already written somewhere else before.

D: Yes.

K: And so then the problem becomes OK, imagine that there are hundreds of thousands of identifiable components of software. How can you organize them in such a way that you know they’re not like sorted alphabetically or (we laugh) or something like that, some useful way to organize them?

D: Yes

K: Well, the object-oriented paradigm provides an answer to that. The object-oriented paradigm provides a way for organizing literally hundreds of thousands of components in a way that you can browse through it and use them and manage that level of complexity and size.

D: Mmm. Mm-hmm Mm-hmm

K: And no other paradigm comes close to that or handles that at all.

D: Yeah

K: So Smalltalk comes in with like a thousand, 1500 built in methods and that’s a lot of functionality. Then on top of that, the third party products you can buy for Smalltalk or C++ or any object-oriented language um just support that same view. You’re just like adding a whole bunch of small software components and it’s like you’ve got like a big library of software components you can then build your code up with so that I can focus on just what’s new in my application, what’s original and use existing components to support everything else. And that’s what I like about the language. So, if I were to think about it abstractly, it’s two things. One, it’s the object-oriented paradigm as a way to model the world and that has something to do with the scale up thing I was talking about.

D: Yes

K: The, because the paradigm follows in some way the way we think about the world, um and its consistency, that that paradigm’s modelling capabilities are supportive of development. And the other is that it supports large amounts of libraries of code. Those are the two things I like about it.
D: And Prolog?

K: Oh, and I like the development environment.

D: OK, Yeah

K: Prolog, I like for AI programming. Um in AI programming you often have to look at very sophisticated algorithms that involve pattern-matching or ideally backtracking or in which you’re developing very rich complex data structures and Prolog as a language supports the creation of algorithms and complex data structures and their manipulation really easily and elegantly. It’s been my experience, I’ve done like say the same functional problem in different languages sometimes. It’s been my experience that in Prolog the solution is like profoundly smaller in terms of lines of source code and has an elegance or a clarity to the design that is greater than any other language that I’ve worked with.

D: Mm-hmm Mm-hmm Just checking how much tape we have ... How much tape is there? Little, OK.

K: So if I want to represent things symbolically.

D: Yes

K: If I want to represent knowledge symbolically or or do reasoning or inference symbolically, Prolog really supports that easily.

D: Mm-hmm Mm-hmm Kay yeah Yeah, It’s been about 50 minutes so I’ve found this gets too tiring as you start to get towards an hour.

K: Yes

D: So um. That’s all the questions I have to ask for now. Is there anything that you’re thinking “Oh I didn’t mean that” or or

K: Ha ha ha

D: or “I need to revise what I said” or anything you want to ask or mention?

K: No

D: OK Good. So what I will do then is transcribe this and next week come back, it was next week we planned I think

K: Let’s look
K: Repeat that

D: Just repeat what you just said.

K: So why I why I chose Computing Science was was for practical considerations, for a job and I thought it sort of met my interests in a vague way, but why I stayed in it is because of um a genuine uh enjoyment with um with programming as an experience. It's creative, it's practical, um I get a gee whiz effect, you know, from doing something with a program. And uh the technology both in software and hardware are changing fast and I find change stimulating.

D: Yeah Yeah

K: And um sort of like related to finding an esthetic beauty in um like in say um a mathematical proof, if I were to, for example, study how a particular natural language processing AI algorithm is done, um, I get a sense of esthetic beauty in the solution and how they've done it.

D: Mm-hmm

K: And I like that and it keeps me in intellectual stimulation.

D: Yeah. Me too.