Department of Computer Science,
University of Otago

Technical Report OUCS-2001-06

My program is correct but it doesn't run: A review of novice programming and a study of an introductory programming paper

Authors:

Anthony Robins, Nathan Rountree, Janet Rountree

Status: Submitted to International Journal of Human-Computer Studies.

Department of Computer Science,
University of Otago, PO Box 56, Dunedin, Otago, New Zealand

http://www.cs.otago.ac.nz/trseries/

My program is correct but it doesn't run: A review of novice programming and a study of an introductory programming paper

Anthony Robins, Janet Rountree, Nathan Rountree

Computer Science,
University of Otago,
P O Box 56, Dunedin,
New Zealand

{anthony, janet, rountree} @cs.otago.ac.nz

Keywords: learning to program, CS1, expert, novice, knowledge, strategy, generation, comprehension, object–oriented, Java

Contents

Abstract

In this paper we review the literature relating to the psychological / educational study of programming. We identify general trends comparing novice and expert programmers, programming knowledge and strategies, program generation and comprehension, and object–oriented vs. procedural programming. Our main focus is on novice programming, characteristic novice behaviour, and topics relating to novice teaching and learning. In the context of this review we briefly describe our own introductory programming paper, COMP103, and note ways in which it addresses key issues relating to programming strategies and mental models of programs. We present data regarding the kinds of problems that students meet while writing their own programs in laboratory sessions. These confirm and extend results noted in the literature. Most novice problems relate to algorithmic complexity in certain language features and in basic program design. A key issue which emerges, but has not previously been well addressed, is the distinction between effective and ineffective novices. What characterizes effective novices? Is it possible to turn ineffective novices into effective ones? We explore these topics and suggest a framework for organizing the knowledge, strategies and models that are involved in programming so as to help diagnose and assist novices.

1.0 Introduction

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

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

Our interest in this research is focused by practical considerations. We teach a computer science introductory programming course, the kind often known as "CS1".

Our goal is to provide the most effective learning environment and experience that we can for the students. Consequently we are interested in understanding the processes of learning and teaching a first programming language. Why is programming hard to learn? What are the cognitive requirements of the task? What can we as teachers do to most effectively support novice programmers? In this paper we describe our current understanding of these issues based on a review of the relevant literature and a study of our own introductory programming paper COMP103.

The literature review (Section 2) provides a general overview of research into programming, identifying several significant trends. Given our underlying interests we focus on novice programmers, exploring their capabilities and typical problems, their characteristic behaviours, and factors relating to course design and teaching. In the context of this review we then (Section 3) briefly describe COMP103. Although the paper addresses several of the points identified in the literature many students still experience significant difficulties. This motivated us to explore further, and we describe a study of the problems encountered by students while writing their own programs during laboratory sessions. (The title of this paper includes a comment from a student gleaned during the course of the study). The study was conducted in 2001 and the course is based on the object–oriented language Java, but in general the results confirm and extend many observations made in studies conducted during the 1980's based on procedural languages. The problems experienced by novices are consistent, fundamental, and not yet addressed by advances in modern language design, textbooks, or (at least in our case) pedagogy. In discussion (Section 4) we suggest that some progress may be made by exploring and contrasting effective and ineffective novices, and in particular focusing on the strategies that they employ. We describe a framework for organizing the knowledge, strategies and models that are involved in programming which may help to diagnose and assist novices. A better understanding of these issues should help us to focus course design and delivery, and better foster novice learning.

2.0 Learning to program

Issues related to programming have been a very active area of research. In this section we briefly identify some of the main themes in the literature, and focus in more detail on the topics of novices, and the teaching and learning of programming.

2.1 Overview

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

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

2.1.1 Experts vs. novices

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

There are many studies of “expert” programmers (although some of these are based on graduate students who are probably only competent or proficient on the

scale noted above). Studies of experts focus in particular on the sophisticated knowledge representations and problem solving strategies that they can employ (see for example Détienne (1990), Gilmore (1990b), Visser & Hoc (1990)). In a survey of program understanding von Mayrhauser & Vans (1994) summarize studies (in particular Guindon (1990)) noting that experts: have efficiently organized and specialized knowledge schemas; organize their knowledge according to functional characteristics such as the nature of the underlying algorithm (rather than superficial details such as language syntax); use both general problem solving strategies (such as divide–and–conquer) and specialized strategies; use specialized schemas and a top–down, breadth–first approach to efficiently decompose and understand programs; and are flexible in their approach to program comprehension and their willingness to abandon questionable hypotheses. Expert knowledge schemas also have associated testing and debugging strategies (Linn & Dalbey, 1989). Rist summarizes many of the advantages of the expert programmer as follows:

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

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

By definition novices do not have many of the strengths of experts. Studies reviewed by Winslow (1996), for example, have concluded that novices are limited to surface and superficially organized knowledge, lack detailed mental models, fail to apply relevant knowledge, and approach programming "line by line" rather than using meaningful program "chunks" or structures. Studies collected in Soloway & Spohrer (1989) outline deficits in novices' understanding of various specific programming language constructs (such as variables, loops, arrays and recursion), note shortcomings in their planning and testing of code, explore more general issues relating to the use of program plans, show how prior knowledge can be a source of errors, and more. Novices are "very local and concrete in their comprehension of programs" (Wiedenbeck, Ramalingam, Sarasamma & Corritore, 1999, p. 278). Since our main interest is in novices and the early stages of learning, we return to this topic in more detail in Section 2.2 below.

2.1.2 Knowledge vs. strategies

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

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

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

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

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

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

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

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

> "Much of the literature concerned with understanding the nature of programming skill has focused explicitly on the declarative aspects of programmers' knowledge. This literature has sought to describe the nature of stereotypical programming knowledge structures and their organization. However, one major limitation of many of these knowledge–based theories is that they often fail to consider the way in which knowledge is used or applied. Another strand of literature is less well represented. This literature deals with the strategic aspects of programming skill and is directed towards an analysis of the strategies commonly employed by programmers in the generation and comprehension of programs." (Davies, 1993, p. 237).

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

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

2.1.3 Comprehension vs. generation

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

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

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

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

Rist (1995) presents a comprehensive model of program generation (see also Rist (1986a, 1986b, 1989, 1990)). Knowledge is represented using nodes in internal memory (working, episodic, and semantic) or external memory (the program specification, notes, or the program itself). A node encodes an "action" that may range from a line of code, to chunks such as loops, to one or more routines of arbitrary size. Nodes are indexed using a tuple of the form <role, goal, object>, for example a read loop could be indexed as <read, stream, – >. Nodes also have four "ports", *:use*, *:make*, *:obey* and *:control*, which allow them to be linked with respect to control flow and data flow. A program is built by starting with a search cue such as <find, average, rainfall>, and retrieving from memory any matching node. Nodes can contain cues, so cues within the newly linked node are then expanded and linked in the same way. Linked systems of code that produce a specific output called plans, and common / useful plans are assumed to be stored by experts as schema–like knowledge structures.

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

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

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

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

2.1.4 Procedural vs. object–oriented

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

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

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

between plans (a fundamental unit of program design, as discussed above) and objects as "orthogonal".

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

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

Two recent studies have explored the problems encountered by novices in detail. Wiedenbeck, Ramalingam, Sarasamma & Corritore (1999) studied the comprehension of procedural and OO programs in subjects in their second semester of study at university. Subjects were learning either Pascal or C++, and were tested on programs written in the language they were learning (but carefully designed so that versions in each language were equivalent). For short programs (one class in C++) there was no significant difference in overall comprehension between languages, though the OO subjects were better specifically at understanding the function of the program. Results were completely different when longer programs (multiple classes) were used, with procedural programmers doing better than OO programmers on all measures. The authors conclude that:

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

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

This view does not support the claim that the OO paradigm is a "natural" way of conceptualizing and modeling real world situations:

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

Similar conclusions are reached by Wiedenbeck & Ramalingam (1999) in a study of C++ students comprehending small programs in C and C++. Once again no difference in overall measures of comprehension were found. Comparing specific measures, however, suggested that subjects tend to develop representations of (small) OO programs that are strong with respect to program function, but weaker with respect to control flow and other program related knowledge. In contrast subjects'

representations of procedural programs were stronger in program related knowledge. Results for the better performing half of subjects were then compared to those of the worse performing group. For the better performing group no difference was found. All differences between the OO and procedural conditions were attributable to the worse performing subjects. Burkhardt, Détienne & Wiedenbeck (1997) propose a theory of OO program comprehension (including the models constructed by programmers and the effect of expertise on the construction of models) within which many of these factors can be explored.

2.1.5 Other

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

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

2.2 Novice programmers

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

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

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

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

2.2.1 The task

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

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

Rogalski and Samurçay summarize the task as follows:

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

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

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

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

2.2.2 Mental models and processes

Writing a program involves maintaining many different kinds of "mental model" (see for example Johnson–Laird (1983)), quite apart from a model / knowledge of the programming language itself.

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

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

> "The notional machine an idealized, conceptual computer whose properties are implied by the constructs in the programming language employed" (du Boulay, O'Shea & Monk, 1989, p. 431).

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

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

> " [a major issue] is the need to present the beginner with some model or description of the machine she or he is learning to operate via the given programming language. It is then possible to relate some of the troublesome hidden side–effects to events happening in the

> model, as it is these hidden, and visually unmarked, actions which often cause problems for beginners. However, inventing a consistent story that describes events at the right level of detail is not easy." (du Boulay, 1989, p. 297 – 298).

Du Boulay, O'Shea & Monk (1989) suggest that to be useful the notional machine should be simple, and supported with some kind of concrete tool which allows the model to be observed. In short, a "glass box" instead of a "black box".

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

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

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

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

Some bugs are minor and can be fixed without change to the program model. In situations where diagnosing a bug exposes a flaw in the underlying model, however, debugging the code may result in major conceptual changes. Pennington & Grabowski (1990) state that diagnosis is the most difficult aspect of debugging, with subsequent corrections being (at least in the case of simple programs where a large re–design is not required) comparatively easier. Gray & Anderson (1987) call alterations to program code "change episodes", and suggest that they are rich in information, helping to reveal the programmers models, goals and planning activities.

2.2.3 Novice capabilities and behavior

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

Some of this rather alarming list relates to aspects of knowledge, and some to strategies. Perkins & Martin (1986) note that "knowing" is not necessarily clear cut, and novices that appear to be lacking in certain knowledge may in fact have learned the required information (e.g. it can be elicited with hints). They characterize knowledge that a student has but fails to use as "fragile". Fragile knowledge may take a number of forms: missing (forgotten), inert (learned but not used), or misplaced (learned but used inappropriately). Strategies can also be fragile, with

students failing to trace / track code even when aware of the process (see also Gilmore (1990b), Davies (1993)).

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

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

programming can also be misleading. Bonar & Soloway (1989) develop this point, exploring the role of existing knowledge (e.g. of step–by–step processes), natural language, and analogies based on these domains as a source of errors. For example some novices expect, based on a natural language interpretation, that the condition in a "while" loop applies continuously rather than being tested once per iteration.

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

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

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

(a) *Summarization problem*. Only the primary function of a plan is considered, implications and secondary aspects may be ignored.

(b) *Optimization problem.* Optimization may be attempted inappropriately.

(c) *Previous–experience problem.* Prior experience may be applied inappropriately.

(d) *Specialization problem.* Abstract plans may not be adapted to specific situations.

(e) *Natural–language problem.* Inappropriate analogies may be drawn from natural language.

(f) *Interpretation problem.* "Implicit specifications" can be left out, or "filled in" only when appropriate plans can be easily retrieved.

(g) *Boundary problem.* When adapting a plan to specific situations boundary points may be set inappropriately.

(h) *Unexpected cases problem.* Uncommon, unlikely, and boundary cases may not be considered.

(i) *Cognitive load problem.* Minor but significant parts of plans may be omitted, or plan interactions overlooked.

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

While specific problem taxonomies could be debated (and are likely influenced by language, task, and context) the underlying claim is important – basic program planning rather than specific language features is the main source of difficulty. A similar conclusion is reached by Winslow (1996), and supported (in some circumstances) by our own investigation as discussed in Section 3.

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

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

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

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

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

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

This would be an interesting topic to pursue further. It may not be the case that the "processing dimension" is any easier to master, but rather that problem modeling and

representation are logically prior, so that novices who are experiencing problems manifest them at that early stage, while those who are working successfully progress through both representation and processing tasks.

2.2.4 Kinds of novice

While much attention has been paid to the study of novices vs. experts, it is clear that it is also useful to explore the topic of novices vs. novices. A group of novices learning to program will typically contain a huge range of backgrounds, abilities, and levels of motivation, and also typically result in a huge range of unsuccessful to successful outcomes. As we might expect, measures of general intelligence are related to success at learning to program (Mayer, Dyck & Vilberg, 1989). As noted above (Section 2.1.5) however, Bishop–Clark (1995) found no clear trends emerging from a review of studies of the effects of cognitive style and personality on programming.

Despite the fact that it is apparently not measured by or significant in the cognitive style and personality tests used so far, different kinds of characteristic behavior are certainly evident when observing novices in the process of writing programs. Perkins, Hancock, Hobbs, Martin & Simmons (1989) distinguish between two main kinds, "stoppers" and "movers". When confronted with a problem or a lack of a clear direction to proceed, stoppers (as the name implies) simply stop. "They appear to abandon all hope of solving the problem on their own" (Perkins *et al.*, 1989, p. 265). Student's attitudes to mistakes / errors are important. Those who are frustrated by or have a negative emotional reaction to errors are likely to become stoppers. Movers are students who keep trying, experimenting, modifying their code. Movers can use feedback about errors effectively, and have the potential to solve the current problem and progress. However, extreme movers, "tinkerers", who are not able to trace / track their program, can be making changes more or less at random, and like stoppers have little effective chance of progressing. Our own observations (Section 4) confirm and extend these general groupings.

2.3 Novice learning and teaching in CS1

2.3.1 Goals and progress

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

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

Given the goals of deep learning an observation that recurs with depressing regularity, both anecdotally and in the literature, is that the average student does not make much progress in an introductory programming course. Exploring roughly semester long courses in middle schools, Linn & Dalbey note that few students get beyond the language features link of the chain, and conclude that "the majority of students made very limited progress in programming" (Linn & Dalbey, 1989, p. 74). A study of students with two years of programming instruction (Kurland, Pea, Clement & Mawby, 1989) concludes on a similar note, that "many students had only a rudimentary understanding of programming". Winslow observes that "One wonders […] about teaching sophisticated material to CS1 students when study after study has shown that they do not understand basic loops..." (Winslow, 1996, p. 21).

Soloway, Ehrlich, Bonar & Greenspan (1983), for example, studied students who had completed a single semester programming paper. When asked to write a loop which calculated an average (excluding a sentinel value signaling the end of input) only 38% were able to complete the task correctly (even when syntax errors were ignored).

2.3.2 Course design and teaching methods

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

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

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

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

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

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

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

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

The laboratory based programming tasks that are part of a typical CS1 course have some pedagogically useful features. Each one can form a "case based" problem solving session. The feedback supplied by compilers and other tools is immediate, consistent, and (ideally) detailed and informative. The reinforcement and encouragement derived from creating a working program can be very powerful. In this context students can work and learn on their own and at their own pace, and "programming can be a rich source of problem–solving experience" (Linn & Dalbey, 1989, p. 78). Working on easily accessible tasks, especially programs with graphical and animated output, can be stimulating and motivating for students. However such tasks should still be based on and emphasize the programming principles that underlie the effects (Kurland, Pea, Clement & Mawby, 1989).

Soloway & Spohrer (1989, p. 417) summarize several suggestions relating to the design of development environments / programming tools that support novices. These include: the use of "graphical languages" to make control flow explicit; a simple underling machine model; short, simple and consistent naming conventions; graphical animation of program states (with no "hidden" actions or states); design

principles based on spatial metaphors; and the gradual withdrawal of initial supports and restrictions. Anderson and colleagues (Anderson, Boyle, Farrell & Reiser, 1987; Anderson, Conrad & Corbett, 1989; Anderson, Boyle, Corbett & Lewis, 1990) have developed an extensive and effective intelligent tutoring system for LISP within the ACT* model of learning and cognition (Anderson, 1983, 1990).

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

2.3.3 Alternative methods and curricula

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

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

> "Explicit naming and teaching of basic schemata [...] may become part of computer programming curricula" (Mayer, 1989, p. 156).
>
> "… students should be made aware of such concepts as goals and plans, and such composition statements as abutment and merging […]. We are suggesting that students be given a whole new vocabulary for learning how to construct programs." (Spohrer & Soloway, 1989, p. 413).

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

Is it effective to teach schemas directly to novices, rather than expect them to emerge from examples and experience? Some general support is provided from a review of mechanisms of skill transfer (see for example Robins (1996)), but transfer

and analogical mechanisms are complex. Deep, structural similarities are often not identified and exploited. While supporting the idea of teaching schemas Perkins *et al.* (1989) also suggest that alternative methods may be more generally effective:

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

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

Problem solving has also been identified as a possible foundation for teaching programming. Deek, Kimmel & McHugh (1998) describe a first year computer science course based on a problem solving model, where language features are introduced only in the context of the students' solutions to specific problems. In this environment students in the problem solving stream generally rated their own abilities and confidence slightly more highly than did students in the control stream (receiving traditional instruction). Students in the problem solving stream also achieved a significantly better grade for the course (with for example an increase from 5% to over 25% of the students attaining "A" grades).

Like schema / pattern based methods the problem solving based approach also appears to have promise. However as noted (Section 2.2.3) by for example Winston (1996) and Rist (1995), problem solving is necessary, but not sufficient, for programming. The main difficulty faced by novices is expressing problem solutions as programs. Clearly the coverage of language features and how to use and combine them must remain a central focus.

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

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

2.4 Summary

The psychological / educational literature relating to programming is large and complex. As we have summarized them the first trend is a distinction between novices and experts, with an emphasis on the many deficits in novice knowledge and strategies. This distinction between knowledge and strategies defines the second trend. An important though ill–defined concept is the schema / plan as the most important building block of programming knowledge. An important but open question is why and how different strategies emerge, and how these are related to underlying knowledge. The third trend is the distinction between program comprehension and generation, with models of the former being particularly numerous. Clearly these tasks are related, with comprehension in particular playing an important role in supporting generation, but there is some suggestion that individuals abilities with respect to these tasks may not be well correlated. The final trend is a recent comparison of OO and procedural programming styles. There is little support for the claim that the OO approach allows for significantly easier modeling of problem domains, with both OO design and traditional procedural factors identified as significant.

In this literature the majority of studies focus on program comprehension, often in experts, and typically based on experimental studies. Our own interest as teachers is in novices, particularly novice program generation, and in the process by which this is taught and learned.

It is clear that novice programmers face a very difficult task. Learning to program involves acquiring complex new knowledge and related strategies and practical skills. Novice programmers must learn to develop models of the problem domain, the notional machine, and the desired program, and also develop tracking and debugging skills so as to model and correct the programs that they develop. Novice knowledge is often fragile – learned in some sense but not applied or misapplied. The most significant difficulties seem to relate not to learning new language features, but to the combination and use of those features, especially the underlying issue of program

design. Different kinds of characteristic novice behavior can be identified, including movers, stoppers, and tinkerers.

With respect to teaching novices in CS1 type courses the goal is to foster deep learning in students. Many students make very little progress in a first programming paper. Explicit attention to program design and relevant mental models may be of assistance. Course designs based on explicitly teaching schemas, problem solving, and mathematical foundations have also been proposed.

3.0 A study of an introductory programming paper

In this section we describe our introductory programming paper, COMP103, and the way in which its design addresses several of the issues that are important for novice learning. We present an initial study of students in COMP103 laboratory based programming work. The results of this study confirm and expand on many of the points identified in the literature review above.

3.1 The design of COMP103

3.1.1 Context

COMP103 is an introductory programming paper following on from a prerequisite COMP101 which teaches general computing concepts and applications. It is taught in the second semester of computer science students' first undergraduate year with a typical enrollment of roughly 400 students, and in a summer school with a typical enrollment of roughly 100 students. For various reasons of departmental curriculum design (and following the usual lively professional debate on such matters) the language taught is Java. The course consists of 24 fifty minute lectures, and 24 two hour laboratory sessions.

We believe that the students who take COMP103 are typical of CS1 students at other universities in similar countries. In general students in our department do well (with some progressing to academia), are highly regarded by employers, and have over the years both won and placed highly in the annual international Association of Computing Machinery (ACM) scholastic programming competition.

COMP103 is a conventional paper in that it uses lectures primarily to present language related knowledge, and laboratory sessions primarily to present practical tasks for which the students write short programs. The guiding principle in the design of the paper was to provide a consistent and well organized "package" to students, so that at any given point the material presented in the text book, the lectures, and the laboratory sessions should be addressing the same topics (in a consistent and well cross referenced way). In order to best achieve these goals we felt that it was useful to base the structure of the course around the order of topics as they were developed

in the text book[4]. The text used to date has been Koffman & Wolz (1999), which COMP103 follows up to and including Chapter 10.

Based on the feedback from independent student reviews of teaching, class representative meetings, and student publications, we can safely say that COMP103 polarizes student opinion. A minority hate the paper, but it is well regarded by the majority. It is seen as very difficult, but also challenging, well organized, and well presented. The pass rate (depending on how it is calculated) is roughly 70%.

3.1.2 Lectures and knowledge

Lectures are used to primarily to present language related knowledge, and to actively engage students with this knowledge so as to foster deep learning. Each lecture is based on a well defined subject such as repetition and loops, and presents this material in the form of "topic cycles". A topic cycle consists of the teacher presenting new material (for example, a description of "for" loops and their use), followed by a short period where students work on an exercise or exercises based on the new material (for example writing a "for" loop to create certain output), followed by the teacher working through a solution to the exercise, then calling for and answering any questions arising from it. In this way students are engaged with the material, encouraged to elaborate on it, and have the opportunity to raise any questions and problems in a timely manner. While it is hard to have a dialogue with 400 students, frequent "show of hands" polls allow the teacher to monitor class progress and fine tune matters such as timing and levels of feedback.

Students are further encouraged to engage with and elaborate course material by the practice of allowing them to take a single sheet of notes into both mid–semester and final exams. With the opportunity to create their own notes students are motivated to actively review the course material, and identify and summarize for themselves the most significant or difficult information[5]. This enhances the formative

[4] Every teacher has their own opinions on the best design / order of presentation for such a paper, and every teacher can make a case for their preferences. From the average student's point of view, however, any advantage arising from a particular teacher's preferred design is likely to be offset by disadvantages if lectures and the text are significantly out of step. We suggest choosing a text which is generally consistent with the teacher's preferences and structuring the course around it. We have received a lot of positive feedback from students in support of this approach, and if nothing else it certainly encourages them to buy and use the text!

[5] We are grateful to a colleague, Paul Werstein, for this excellent suggestion.

potential of the examination based assessment. The exams are based on both multi–choice and short answer questions.

One of the main goals of the lecture material is to present visual models and animations (which are easy to construct using presentation software and data projected slides) of running Java programs. Right from the start simplified models of Java programs are consistently presented, and animations are used to illustrate such topics as the behavior of loops, the creation and manipulation of objects, and the behavior of references (pointers) and reference data types. A typical slide from a typical animation is shown in Figure 1. As would be predicted by the theory reviewed above and is supported student feedback, these models were very useful. In retrospect, however, we failed to sufficiently address the concept of the underlying notional machine itself.

The textbook, and consequently the course as a whole, adopts an "objects early" approach. OO concepts are introduced from the start, and initial program designs are based on an application class and a single support class. Conditionals, loops, arrays and the like are introduced after this foundation has been established, with applets and graphical user interfaces (GUIs) introduced towards the end of the course.

3.1.3 Laboratory sessions and strategy

Laboratory sessions were used primarily to present practical tasks for which the students write short programs. In practical terms the lab provides roughly 40 iMac workstations running the CodeWarrior[6] development environment (of which only a subset of the functionality was used). Students are supported by between 3 and 5 demonstrators (teaching assistants), who are typically senior students in the department. Each laboratory exercise contributes some assessment weighting (typically 1%) to the course, and there are 24 two hour lab sessions in all. Students were expected to complete a small number of revision questions set out in the lab book before each lab session. They were also expected to prepare for the programming task and plan the program to be written during the assigned lab time.

The preparatory program planning was intended to be a central part of the attention to programming strategy in COMP103. The Koffman & Wolz (1999) textbook was adopted in part because it presents a useful OO "software development method", and

[6] CodeWarrior is a registered trademark of Metrowerks Inc.

ms2.setAll(4, 5, 6);

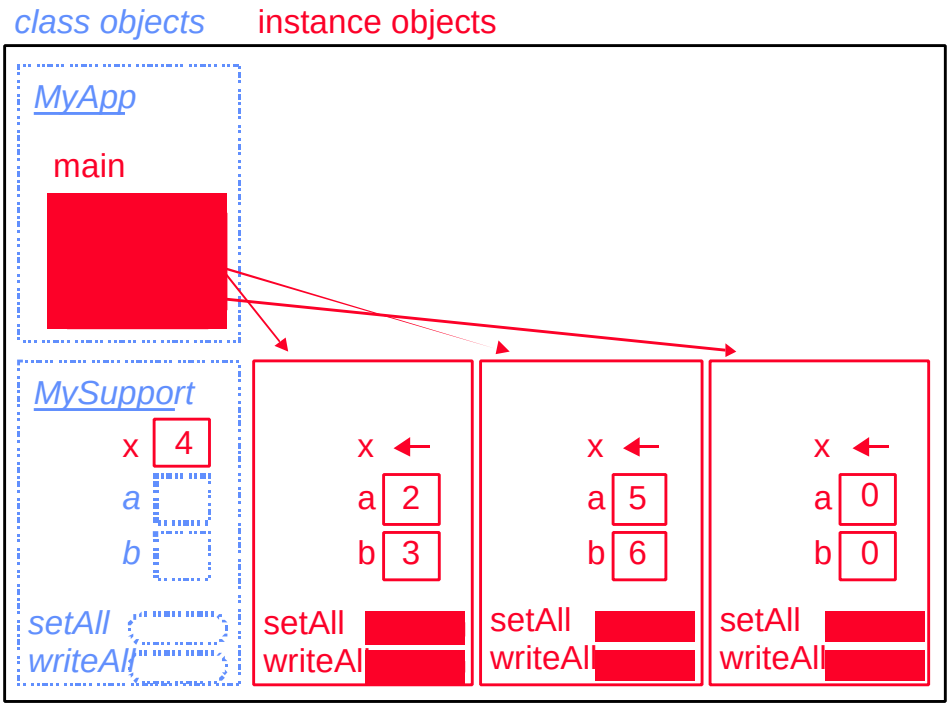


Figure 1: Slide from a typical program model animation sequence

The program consists of an application class which creates three instances (ms1, ms2 and ms3) of a support class. The support class has one class (static) data field *x*, two instance data fields *a* and *b*, and methods for setting and writing out the values of the data fields. The animation consists of a sequence of slides showing the state of the program as each of the instance objects are created in turn, then the setAll method is called on each instance object to set value of its data fields, then the writeAll method is called on each object. The particular slide shown above is from a point in the sequence where the setAll method has just been called on the second instance object ms2 (as indicated by the line of code at the top left). The purpose of the animation sequence is to illustrate the difference in the behavior of the class vs. the instance data fields.

uses it consistently in the development and presentation of examples. The software development method consists of a series of steps: specification, analysis, design, implementation, testing and maintenance. The design step involves top–down planning of the classes to be used and the algorithms to be implemented by each class. While this goes some way to addressing the central issue of program design, in retrospect probably too little attention is paid to aspects such as control flow and data flow that have been identified as important in supporting the OO model. Nevertheless the method provides some instruction in design and programming strategy, and a framework on which further discussion can be based.

Programming strategy was further addressed in COMP103 with highly structured task descriptions in the lab workbook, notes during these descriptions pointing out strategic aspects of the process, and short appendices in the lab workbook that describe problem solving methods and OO design considerations. In particular, the highly structured task descriptions marked a significant change from the briefer task specifications used in earlier versions of the course. The descriptions provided a lot of guidance for both the process of understanding the task and the process of creating the program. This approach was adopted partly in recognition of enormous difficulties that novice programmers face, and partly because of the our perspective on the purpose of the lab work. Effective problem solving and programming is based on a foundation of relevant knowledge and strategies, and we saw the purpose of the lab work as building up that foundation, rather than "testing" the students or "throwing them in at the deep end". Confident students were welcome to skip the detailed guidance and attempt the assigned task in their own way.

3.1.4 Summary

In short, we believe that COMP103 has been designed and delivered in such a way that it explicitly addresses many of the recommendations and concerns about teaching and supporting the learning of novice programmers identified in the literature. We believe that it is a good example of a conventionally structured course.

3.2 The study

3.2.1 Background

For the last two years we have been collecting data from students in COMP103. This includes "before" and "after" survey questionnaires, detailed case studies of individual students, and notes and checklists regarding problems encountered by students during laboratory work. An analysis of the questionnaire data as a predictor of success and failure in COMP103 can be found in Rountree, Rountree & Robins (2001).

Much of the data has been collected from students working in the laboratory sessions. Demonstrators (teaching assistants) are called (using an automated call / queuing system) to assist students when they encounter problems. The research assistant works like a demonstrator in the lab, and thus gets called to assist with a randomly selected sample of student problems. Notes can be made regarding the nature of the problem, the advice given, and subsequent progress. Occasionally samples of code are collected. In this way a lot of detailed data and experience regarding actual student problems and behaviors in a naturalistic setting can be accumulated. This includes information rich change episodes (see Section 2.2.2). One specific disadvantage of this approach, however, is that it captures much more information about ineffective novices than effective ones (who tend to call for assistance much less often).

In general terms this methodology has the typical advantages and disadvantages of naturalistic observation – see for example Sheil (1981), Gilmore (1990a). It lacks the formal rigor of an experimental study, but is high in ecological validity. For our purposes, the qualitative and practical understanding and improvement of learning in COMP103, the observation of behavior in the actual laboratory setting is the obvious way to proceed. A major factor constraining the study is that any process adopted must be practical, and not impact on teaching and learning in the labs. In particular the demonstrators' job is very difficult and can be stressful, no major or time consuming change in their routine was considered. Demonstrators are the true "front line" teachers in COMP103.

3.2.2 Method

The data described in this paper is based on a checklist of the most common problems encountered by students. The checklist was compiled after the first year of the study and consists of 36 problem types covering the use of the Macintosh, the use of the CodeWarrior development environment, understanding of the problem, algorithm and program design, and a range of language based issues (including some which are specific to Java). A description of the checklist problem types is provided in Appendix A.

Checklist data was gathered from the 2001 "summer school" version of COMP103, with an initial enrolment of roughly 100 students. After two introductory labs every demonstrator in every lab session used a copy of the checklist. After assisting each student the demonstrator entered a tally mark on their check list for any problem type about which she/he had given advice. In this way a lab by lab count of the number of times each problem type was encountered could be compiled.

Our initial expectation was that the problems encountered in each lab would be largely associated with the new language features that were introduced, with perhaps a few specific problem types being unusually common or persistent. Such data would be very useful for identifying the most problematic aspects of the language, and tailoring the course to specifically address these issues[7].

There are many problems with the checklist methodology. In any given lab session the majority of students are working on the currently assigned lab exercise, but some may be working on other exercises (past or future) or their own projects. Thus the problems tallied are not all related to the current exercise. Extrinsic factors such as impending holidays or exams influenced lab attendance, hence some sessions have unusually low or high overall problem tallies. After assisting a student the demonstrator has to judge which problem types to check. This introduces obvious individual differences in the way student problems are classified. We tried to address this point in a number of ways. Firstly, demonstrators were asked to check only problem types about which they had given specific advice (rather than, for example, any other problems that they thought the student may be experiencing). Secondly, we tried to reduce individual differences by short training sessions in the use of the checklist and the problem types, and by monitoring of and feedback regarding the

[7] Such a process was in fact carried out, and a number of problem specific "help packs" were produced.

process. Here the research assistant served as a consistent point of reference, giving advice on difficult cases and generally assisting and coordinating the demonstrators.

In short, the lab based problem tallies should not be interpreted as fine grained quantitative data. We believe, however, that they provide an interesting perspective on activity in COMP103 laboratory sessions, and are useful for identifying general qualitative trends.

3.3 Results

3.3.1 Lab based problem tallies

Detailed results for selected laboratory sessions are presented in Appendix B[8]. Descriptions include a brief outline of the laboratory exercise, a note of the new language features introduced, an indication of the expected checklist problem types, and a graph of the actual problem type tallies collected for that lab session. This data is useful mainly at the level of the general trends that emerge.

3.3.2 Trends

As predicted from the literature review the basic design and planning of a working program is a significant and ongoing problem. This is true even when the problem itself is understood – compare problem types P2 and P3 (Appendix A) for Labs 18 and 23 (Appendix B). Basic design problems can be far more significant than problems with any particular language feature (see P3 in Lab 14).

These observations need to be qualified however. Basic design is identified as a frequent problem only once the required programs reach a certain level of algorithmic / procedural complexity. In our study this occurred in Lab 14. Students had met conditionals (if) and loops in previous labs, but Lab 14 was the first in which they were required to combine and use both features in order to solve the problem. Subsequently, basic design was identified as a frequent problem in all labs (which had a similar level of algorithmic complexity). Note that it is algorithmic, rather than OO complexity, which appears to be the trigger. With the "objects early" approach of the course students had already done lab work using inheritance and the creation of

[8] Details for all labs can be obtained from the first author.

multiple instance objects without encountering the design problems seen from Lab 14 on.

Some specific language features / constructs also cause problems when they are introduced. This includes, for example, the use of parameters and returned values (Lab 7), and notably the introduction of loops (Lab 14) and arrays (Lab 16). Other language features do not appear to trigger large numbers of problems. These include simple string processing (Lab 7), conditionals (Lab 14), I/O from files (Lab 18), and to some extent the Abstract Windowing Toolkit (AWT) and Java event model (Lab 23). Once again the critical factor seems to be algorithmic complexity (or perhaps in the case of arrays, the capacity to support it). While some language features are complex in the sense of involving a lot of detail, they do not cause the same number of problems as are caused by algorithmic complexity. Mirroring the case for general design, problems associated with specific OO language features are far less prevalent than problems associated with algorithmically complex features.

However, observations regarding specific language features also need to be taken with a healthy grain of salt. In some cases students appear coping with new features without difficulty, but may in fact be "just following instructions", and encounter problems later on when they are expected to use these features more creatively. Inheritance, for example, is first used in Lab 9 without a single recorded problem, but is a major problem identified in Lab 18. To a certain extent, where we observe problems with language features may be where we have invited them by expecting those features to be used in ways that demonstrate deep learning. While we believe that lab exercises must be well specified to support novice learning, such considerations are certainly a complicating factor.

Some apparently basic issues cause continual problems for some students. Even in later labs problems were observed with typos and spelling, brackets and missing semicolons, and the use of CodeWarrior. An uncommon but persistent underlying problem was students who were working on programs without a clear understanding of the problem they were trying to solve.

Finally, at least with our methodology, problems can manifest themselves at unexpected focal points (Section 2.2.3). While it is only an intuitive observation, a problem using constructors (P23, see Labs 9, 16, 18) often seemed to be the tip of an iceberg of conceptual muddle regarding objects, class, instance, reference types, and

program design in general. A constructor is a simple enough thing unless you do not understand what it is you are constructing, and why.

3.3.3 Other observations

Some observations not specifically related to the problem checklist data are worth briefly noting. Programming lab work shows up a huge range of abilities and progress. Some students find the material easy and enjoyable, some find it impossible, and for the larger group in between attitude and application have an enormous effect on the outcome. The graphical work introduced in the latter lab sessions was very popular, motivating some students to put extra work into making their programs more exciting.

Students were encouraged to use the software design method for every lab, and plan their program before the lab session. The majority did this initially, but use of the method fell off steadily. However, those that did continue to use the method consistently tended to do very well, progressing through most labs with fewer problems than other students.

Finally, an unexpected and subtle “early warning” sign was tentatively identified. Students who had the most basic problems with planning and design also seemed to have trouble naming things (classes, methods, data fields, variables). In other words, naming things so as to capture their role / function seems to be another focal point, and problems with naming may be indicative of underlying conceptual confusion.

4.0 Discussion

Our results confirm and in some cases extend earlier observations relating both to novice programming in general and novice programming in the OO paradigm.

In the absence of standardized measures and quantitative data it is very hard to compare different studies, and the impact (or otherwise) of various manipulations of curriculum, course design or teaching methods. It is clear, however, that many of the problems and factors that were identified in studies conducted on procedural languages during the 1980's are still significant in a study of an OO language conducted in 2001. This is the case even in the context of a course designed in such a way that it explicitly addresses many of the issues earlier identified as important (exploring strategies as well as knowledge, making extensive use of graphical program models and animations, actively engaging students, and so on). While novice problems may well have been reduced in COMP103, they have certainly not been resolved. The problems that novice programmers experience are consistent and fundamental.

How can we improve this situation? In this section we identify the crucial question as being the distinction not between experts and novices, but between effective and ineffective novices. We suggest that the essence of this distinction lies not in knowledge, but in strategies, and we propose a framework within which novice problems can be diagnosed and hopefully addressed.

4.1 Kinds of novice

Many of the general observations arising from the study relate to kinds of novice / student behavior. With only slight extension the simple categories proposed by Perkins *et al.* (1989) (see Section 2.2.4) are useful, and we characterize novices as either effective (planners, movers) or ineffective (tinkerers, stoppers). These categories should be interpreted as describing typical behavior rather than immutable absolutes. Everyone has good days and bad days!

The majority of students work effectively. Planners have a clear conceptual framework, plan programs carefully, and work on them systematically. Movers typically have a partial plan that they refine and expand as they develop their code.

They experiment, modify their programs, and can use feedback such as error messages effectively.

A smaller, but nonetheless sizable group of students work ineffectively. Tinkerers are extreme movers with rudimentary plans. They change their code continuously, guided more by optimism than design. They do not use feedback effectively, and may even actively resist suggestions or offers of help. Stoppers frequently reach a point in planning or implementation where they cannot proceed without assistance. They do not use feedback or other sources of information effectively. In some cases they may have an emotional reaction to setbacks and errors.

From our point of view the distinction between effective and ineffective novices is much more important than the one between experts and novices which has received so much attention in the literature. What underlying properties make a novice an effective novice? How can we best turn ineffective novices into effective ones? A deeper understanding of effective novices and a more sophisticated categorization and analysis of ineffective novices is required.

Although they are of less significance, for completeness we note that two other small but well defined groups were identified in the study. These are characterized by approaches which are either derivative or inappropriate. Some students try to make progress by deriving program solutions from elsewhere. This usually means trying to monopolize demonstrators for continuous “assistance”, and may involve copying a related program from elsewhere (usually the textbook) as a starting point. A second small group engage in behavior which is clearly inappropriate, copying work (particularly programs) from other students and presenting it as their own. For obvious reasons neither of these groups meets with conspicuous success. Inappropriate behavior is often identified either because such students fail to recognize changes in the lab book that occur from year to year, or by administrative programs specifically designed to check for duplication. Similar inappropriate behavior has been reported in high school students (Kurland, Pea, Clement & Mawby, 1989), and is probably present at some level in most similar courses.

4.2 Knowledge, strategies, and effective teaching and learning

While it is interesting to describe different kinds of novice behaviour, it is even more useful to understand the underlying factors (knowledge, strategies, abilities and attitudes) which result in these behaviour types. We suggest that the most significant differences between effective and ineffective novices relate to strategies rather than knowledge.

Novices are assumed to have uniformly low language related knowledge, and are required to develop this gradually. Less attention is usually paid to their wide range of strategies, which they are required to use in the laboratory (and which have a great effect on progress) from day one. The elements of language knowledge are readily available in the text book, lab book, lecture notes, online help and tutorials, and feedback from the compiler. Such resources are not useful, however, without the will and the strategies to access and enable the contents. Conversely, a novice with effective strategies could in principle (and occasionally does in fact) use such resources to teach themselves to program without any formal instruction in language knowledge at all.

Others have also suggested that strategies are central. Perkins *et al.* note that "certain broad attitudes and conducts" characterize unsuccessful novices:

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

These are all deficits in strategy rather than knowledge. Davies states that:

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

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

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

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

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

What are the implications of this view for teachers? Firstly, we suggest that programming strategies should receive more and more explicit attention in introductory programming courses. For example, as noted above (Section 2.3.2), the strategies that go into creating a program are not usually visible in the final product. One way to address this would be to introduce many examples of programs as they are being developed (perhaps 'live" in lectures), discussing the strategies used as part of this process. Secondly, we suggest that introductory programming courses should include as a specific element, especially in their early stages, explicitly focusing on teaching / learning how to be an effective novice. Effective novices learn to program. Rather than focusing exclusively on the difficult end product of programming knowledge, it may be more effective to focus at least in part on the starting point of being an effective novice.

Our experience in COMP103 suggests that it is not enough to simply supply (and encourage the use of) information about problem solving strategies, program design factors, and a software development method. What would a more extensive attempt entail? Firstly, we need to know more about what characterizes effective novices (as compared to experts or ineffective novices). It will be useful to explore their strategies for problem solving, planning algorithms, creating program code, and for accessing and using available sources of language related knowledge – much specific detail is required. Secondly, we need to know more about how to foster these attributes in all novices through course design and delivery. Finally, we need to motivate students, engage them in the process, and make them want to learn to be effective programmers.

It is likely that the most successful way of supporting novices is specific individual diagnosis and assistance. The range of potentially relevant factors includes motivation, confidence or emotional responses, general or specific strategic deficits, general or specific knowledge deficits, and general or specific deficits in mental models. Probably the most significant factor influencing the success of programming instruction will be the extent to which high quality personally tailored assistance is available.

4.3 A framework

Combining perspectives from the literature review and the results of our own study, our current view of the issues that are important to consider with respect to programming is captured by the "programming framework" shown in Figure 2. This framework highlights the major dimensions of knowledge, strategies, and mental models over the phases of designing, creating and evaluating a program. The categories / cells of the framework should be considered as "fuzzy", overlapping and blending with each other.

This framework has helped us to clarify and focus on certain issues and their relationship. For example, it is clear that most textbooks and conventional course designs focus on the Knowledge:Creation category (which encompasses language related knowledge). We suggest placing most emphasis on the Strategies:Creation category, and exploring the way it is supported by the rest of the framework. There are many open questions. Why do many novices, even when aware of the techniques and encouraged to use them, fail to plan their programs? What are the main reasons why many students become so consistently stuck, and can these be diagnosed and addressed? Are strategy deficits generic or related to an inability to construct or maintain a mental model of the program? What kind of support will best address the needs of each kind of novice? How can we present language related knowledge so as to best develop and foster appropriate strategies and models? Perhaps one of the most important aspects to be explored is why relevant knowledge and strategies are often known but not used (see the discussion of fragile knowledge, Section 2.2.3).

The framework may be useful for pedagogical purposes, to serve as a starting point for exploring the characteristics of both effective and ineffective novices, or as the basis for a tool for diagnosing and assisting individual students. For this latter purpose in particular, any diagnostic tool to be used by demonstrators in an actual laboratory situation will need to be rich enough to be useful, but simple enough to be manageable.

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

Figure 2: A programming framework

This framework organizes topics relating to programming, particularly program generation. It should be read mainly by columns, i.e. knowledge of planning methods (required to design a program), knowledge of a language (required to create a program), knowledge of debugging tools (required to evaluate a program), and so on.

5.0 Summary

Novice programmers face a very difficult task. Learning to program involves acquiring complex new knowledge, and related strategies for designing, coding, and tracking a program. An important and open question is how and why certain strategies emerge, and the nature of their relationship to representations of programming / language knowledge. Novice knowledge and strategies are often fragile – learned in some sense but misapplied, or not applied at all. Writing a program also involves a mental model of the notional machine, and requires developing models of the problem domain and the program in various states. The most significant difficulties seem to relate not to individual language features, but to their combination and use, especially the underlying issue of program design.

Our own study of novice learning and program generation in COMP103 confirms and in some cases extends observations noted in the literature review. Once programs reach a certain level of algorithmic complexity basic design problems become significant, and some times dominant. Specific language features can also cause problems when they are introduced. Features that introduce algorithmic complexity (particularly loops and arrays) cause significantly greater problems than either those introducing OO structure (such as inheritance) or involving large amounts of detail (such as the Java AWT). Some interesting "focal point' problems relating to constructors and to basic naming issues were observed.

We suggest that, from a teacher's point of view, the distinction (seldom addressed in the literature) between effective and ineffective novices is highly significant. Novices come with a wide range of backgrounds, abilities, and levels of motivation. Can we understand, diagnose, and assist ineffective novices, helping them to become effective? As a starting point we have proposed the very general categories (based on Perkins *et al.* (1989)) of planners and movers (effective), and stoppers and tinkerers (ineffective). More specific detail is required however. In particular, we suggest that the most important differences between novices relate to their strategies rather than their knowledge. Language related knowledge is available from many sources, and courses and textbooks are designed to introduce this knowledge in a structured way. The strategies for accessing this knowledge and applying it to program comprehension and generation, however, are crucial to the learning outcome, but typically receive much less attention. What are the strategies employed by effective novices, how do they relate to their knowledge and their relevant mental models, and can these strategies be taught to ineffective novices?

While we are far from answering such questions, we have proposed a programming framework which helps to set many of the issues into an explicit context. This framework emphasises the dimensions of knowledge, strategies, and models, over the phases of designing, creating, and evaluating a program. We assume that for large classes the most practical way to provide individual attention and assistance to students is via well trained and well supported demonstrators (teaching assistants). Consequently, our current goal is to develop the programming framework into a tool which is both rich enough and simple enough to be useful to demonstrators in diagnosing and assisting students.

In future work we intend to further explore these issues and to focus on novice strategies. We also intend broaden our literature review to include an evaluation of alternative curricula and methods based on schemas and patterns, problem solving, or mathematically based approaches to teaching programming. A number of other specific questions have emerged from the current review. What is the relationship between the ability to generate and the ability to comprehend a program? Is it really the case that formulating the representations used by a program is harder than designing the necessary processing / algorithms? Is diagnosis the most difficult aspect of debugging, and if so how can we better support it? What can we learn from the change episodes that occur as a part of debugging and program design? Finally, of course, the underlying issue is how best to use the answers to such questions to better teach and foster the learning of novice programmers.

Acknowledgments

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

References

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

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

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

Anderson, J R (2000) *Cognitive Psychology and Its Implications: Fifth Edition*. New York: Worth Publishing.

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

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

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

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

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

Bonar J & Soloway E (1989) Preprogramming knowledge: a major source of misconceptions in novice programmers. In Soloway & Spohrer (1989), 324 – 353.

Brooks F P Jr, (1995) *The Mythical Man–Month: Essays on Software Engineering Anniversary Edition.* Reading MA: Addison–Wesley.

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

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

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

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

Burton P (1998) Kinds of language, kinds of learning. *ACM SIGPLAN Notices, 33(4),* 53 – 61.

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

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

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

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

Détienne F (1990) Expert programming knowledge: a schema based approach. In Hoc *et al.* (1990), 205 – 222.

Dijkstra E W (1989) On the cruelty of really teaching computer science. *Communications of the ACM, 32 (12),* 1398 – 1404.

Dreyfus H & Dreyfus S (1986) *Mind Over Machine : The Power of Human Intuition and Expertise in the Era of the Computer.* New York: Free Press.

du Boulay B (1989) Some difficulties of learning to program. In Soloway & Spohrer (1989), 283 – 299.

du Boulay B, O'Shea T & Monk J (1989) The black box inside the glass box: presenting computing concepts to novices. In Soloway & Spohrer (1989), 431 – 446.

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

Gilmore D J (1990a) Methodological issues in the study of programming. In Hoc *et al.* (1990), 83 – 98.

Gilmore D J (1990b) Expert programming knowledge: a strategic approach. In Hoc *et al.* (1990), 223 – 234.

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

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

Green T R G (1990) Programming languages as information structures. In Hoc *et al.* (1990), 117 – 137.

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

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

Hoc J M (1989) Do we really have conditional statements in our brains? In Soloway & Spohrer (1989), 179 – 190.

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

Hoc J M & Nguyen–Xuan A (1990) Language semantics, mental models and analogy. In Hoc *et al.* (1990), 139 – 156.

Humphrey W S (1999) *Introduction to the Team Software Process.* Reading MA: Addison–Wesley Longman Inc.

Johnson–Laird P N (1983) *Mental Models.* Cambridge: Cambridge University Press.

Kahney H (1989) What do novice programmers know about recursion? In Soloway & Spohrer (1989), 209 – 228.

Kessler C M & Anderson J R (1989) Learning flow of control: recursive and iterative procedures. In Soloway & Spohrer (1989), 229 – 260.

Koffman E & Wolz U (1999) *Problem Solving With Java.* Reading MA: Addison–Wesley.

Kurland D M, Pea R D, Clement C & Mawby R (1989) A study of the development of programming ability and thinking skills in high school students. In Soloway & Spohrer (1989), 83 – 112.

Linn M C & Dalbey J (1989) Cognitive consequences of programming instruction. In Soloway & Spohrer (1989), 57 – 81.

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

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

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

Mayer R E (1989) The psychology of how novices learn computer programming. In Soloway & Spohrer (1989), 129 – 159.

Mayer R E, Dyck J L & Vilberg W (1989) Learning to program and learning to think: what's the connection? In Soloway & Spohrer (1989), 113 – 124.

Mendelsohn P, Green T R G & Brna P (1990) Programming languages in education: the search for an easy start. In Hoc *et al*. (1990), 175 – 199.

Mills H D (1993) Zero Defect Software: Cleanroom Engineering. *Advances in Computers, 36,* 1 – 41.

Ormerod T (1990) Human cognition and programming. In Hoc *et al*. (1990), 63 – 82.

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

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

Pennington N & Grabowski B (1990) The tasks of programming. In Hoc *et al.* (1990), 45 – 62.

Perkins D N, Hancock C, Hobbs R, Martin F & Simmons R (1989) Conditions of learning in novice programmers. In Soloway & Spohrer (1989), 261 – 279.

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

Perlis A, Sayward F & Shaw M (1981) *Software Metrics: An Analysis and Evaluation.* Cambridge MA: MIT Press.

Proulx V K (2000) Programming patterns and design patterns in the introductory computer science course. *Proceedings of the Thirty–First SIGCSE Technical Symposium on Computer Science Education,* 80 – 84. New York: ACM Press.

Ramsden P (1992) *Learning to Teach in Higher Education.* London: Routledge.

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

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

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

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

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

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

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

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

Rogalski J & Samurçay R (1990) Acquisition of programming knowledge and skills. In Hoc *et al.* (1990), 157 – 174.

Rountree N, Rountree J & Robins A (2001) Identifying the danger zones: predictors of success and failure in a CS1 course. *Technical Report OUCS–2001–06,* Computer Science, The University of Otago, New Zealand. Submitted to *SIGCSE Bulletin*.

Sackman H (1970) *Man–computer problem solving.* Princeton N J: Auerbach.

Samurçay R (1989) The concept of variable in programming: its meaning and use in problem solving by novice programmers. In Soloway & Spohrer (1989), 161 – 178.

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

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

Sheil B A (1980) Teaching procedural literacy. *Proc. ACM Annual Conference,* 125 – 126.

Soloway E, Bonar J & Ehrlich K (1989) Cognitive strategies and looping constructs. In Soloway & Spohrer (1989), 191 – 207.

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

Soloway E & Ehrlich K (1984) Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering, SE–10(5),* 595 – 609.

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

Soloway E & Spohrer J C (Eds.) (1989) *Studying the Novice Programmer.* Hillsdale N J: Lawrence Erlbaum.

Spohrer J C & Soloway E (1989) Novice mistakes: Are the folk wisdoms correct? In Soloway & Spohrer (1989), 401 – 416.

Spohrer J C, Soloway E & Pope E (1989) A goal/plan analysis of buggy Pascal programs. In Soloway & Spohrer (1989), 355 – 399.

van Dijk T A & Kintsch W (1983) *Strategies of Discourse Comprehension.* New York: Academic.

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

Visser W & Hoc J M (1990) Expert software design strategies. In Hoc *et al*. (1990), 235 – 250.

von Mayrhauser A & Vans A M (1994) *Program Understanding – A Survey.* Technical Report CS–94–120, Department of Computer science, Colorado State University.

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

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

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

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

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

Winslow L E (1996) Programming pedagogy -- A psychological overview. *SIGCSE Bulletin, 28 (3),* 17 – 22.

Appendix A: Demonstrators' checklist

The checklist used by demonstrators in the laboratory sessions consisted of 36 problem type headings on a single sheet. The problem type numbers and headings are shown below, along with a brief definition of what kind of problem each is supposed to cover. These definitions were discussed with the demonstrators and the use of the checklist monitored by the research assistant, as described in the main text.

Problem types P1 and P2 are about basic tools. P3, P4, and P5 are about understanding the task, and planning the algorithm and program that will implement a solution. P6 – P36 are "language related" problem types.

P1 Basic getting around
Using the Macintosh; saving files; copying files; knowing where files are; creating multiple files; reading instructions.

P2 CodeWarrior & IE
Setting Java target; files open but not project; files not linked to project; files in different folders; quitting JBoundApp; trying link file already in project; debugger turned on; disassembling code; trouble with IE (Internet Explorer, used to run applets).

P3 Understanding the problem
Difficulties understanding the task and how to solve it.

P4 Problem into algorithm
Understand the task & solution but can't translate that into an algorithm / code.

P5 General program design
Questions like "I don't know where to start"; "what do I call the class"; copying text book examples or any previous lab when the pattern is inappropriate for the new task.

P6 Program flow
The role of the main method; writing methods and not calling them; unintentional recursion.

P7 Program structure
Code outside of methods; methods that overlap; data fields outside the class; basically not knowing where stuff goes.

P8 OO concepts
Problems with concepts of state verses behavior; what data fields are for; why should have an object.

P9 Tidiness & comments
Messy layout; not following language conventions; incorrect or misleading comments; poorly chosen names.

P10 Braces brackets & semi–colons

Problems with { }, incorrect number or position for ( ) or missing ;.

P11 Typos and spelling
Typos and spelling errors; mixing or misusing uppercase and lowercase.

P 12 Differing names (files)
File name different from class; name used when method called not name of method.

P13 Expressions & calculations
Problems with the math; representing math as Java code; precedence.

P14 Initialization
Not realizing data fields are initialized to 0 and method variables, arrays aren't; null references; arrays of null references.

P15 Return types
Mismatch of declared and actual return types, e.g. public void getTime() {return time}

P16 Arguments and parameters
Misunderstood purpose, incorrect number, type mismatch.

P17 Visibility and scope
Variables not declared; re–declared; trying use method variable as if it were a data attribute; methods given visibility; not understanding effect of visibility; incorrect visibility.

P18 Misuse of data type & casting
Problems with data types and casting, e.g. declared as an int and trying use as a double or char as string. Incorrect casts.

P19 Data fields
See P17. Other problems with data fields, unnecessary extras or messy use of them.

P20 Methods variables
See P17. Other problems with method variables, unnecessary extras or messy use of them.

P21 Accessors
Incorrect use or misunderstood purpose.

P22 Modifiers
Incorrect use or misunderstood purpose.

P23 Constructors
Incorrect use or misunderstood purpose.

P24 Class versus instance

Problems with class, instance, static. For example, Lecturer anthony = new Lecturer(); Lecturer.talk();. Trying to call instance methods (non–static) from main (static); not understanding that an applet is an instance.

P25 Reference types
Misunderstanding handles/references/pointers; a = b; for primitive verses reference.

P26 Inheritance (overriding, super)
Problems with inheritance; use of super( ); unrecognized overriding.

P27 Conditions & booleans
Misunderstood purpose; malformed expression

P28 Loops
Misunderstood purpose; not written or used correctly

P29 Selections
Misunderstood purpose; incorrect use e.g. if if {if else else}...

P30 Arrays & vectors
Misunderstood or confusion between. Difference between index and array size, index and contents; realizing they can hold objects; questions about creating them.

P31 Formatting (strings & output)
Problems formatting output, e.g. use of /n or +. Other problems with strings.

P32 Import statements
Missing import statements; realizing importing java.awt won't include event classes.

P33 I/O & exceptions & files
Problems with I/O and exceptions; opening and using files.

P34 GUI mechanics (exist and don't call)
Problems using GUI classes and features, e.g. created object buttonPanel of type Panel and have neglected add(buttonPanel); problems with GUI layout and connections.

P35 Implementing interface (methods required to create)
Problems with interfaces, e.g. forgetting init() or run() or actionPerformed(); forgetting "implements ActionListener".

P36 Java model
Understanding the Java model, especially with respect to GUIs, applets and the applet context. The AWT event model; what the browser runs for you; what graphics object model does (such as need to repaint the canvas object in order to change background colors).

Appendix B: Results for typical laboratory sessions

Lab 4

The first two labs were based on exploring the CodeWarrior development environment and basic concepts, so this was the second "programming" lab. Like other early sessions, this lab recommends the same simple program structure used by all early examples in the text and the lectures. This consists of an application class and a single support class. The main method creates a single instance object (of type support class), and calls methods on the object to implement the desired algorithm. Input and output are handled by methods in the simpleIO package that came with the textbook (Koffman & Wolz, 1999).

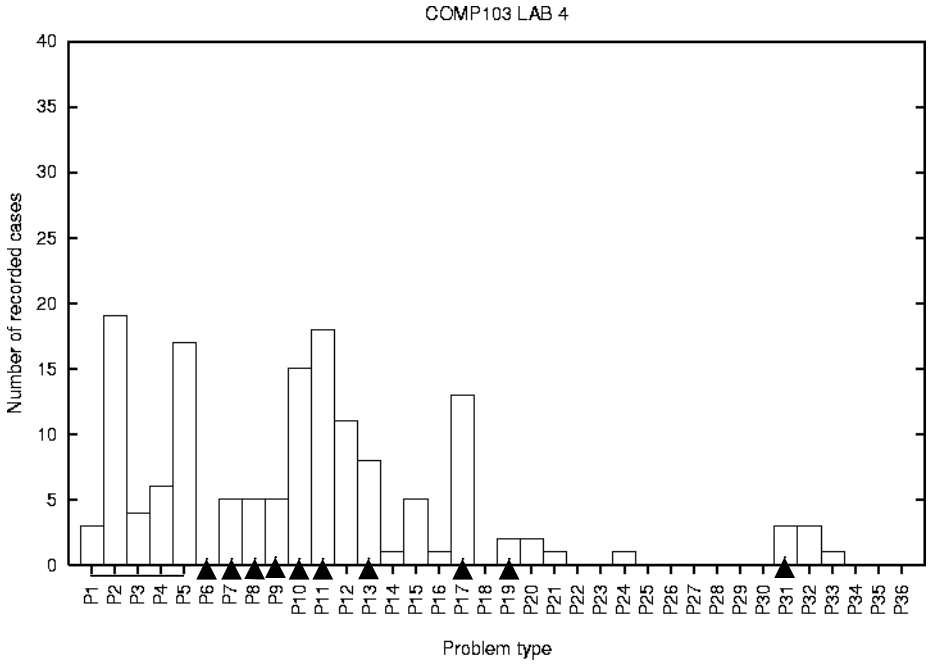
Task:

Read in two numbers. Output the sum, difference, product and quotient. An outline defining classes and methods (but not data fields or method bodies) with was supplied to those who wanted it.

New language features:

In this early lab all aspects of basic Java program structure and function are unfamiliar. The outline provided class and method skeletons, but we still expected basic and planning / design related problems. The task involved data fields, arithmetic, and simple formatting of output. The language related problem types (P6 – P36, see Appendix A) expected are marked with a triangle on the X axis of the graph of problems actually observed. Basic and planning / design related problems (P1 – P5, see Appendix A) are marked with a line.

Problems observed:



Lab 7

This lab is based on an extensive discussion of the ways that methods can "input" (parameters, access data fields) and "output" (return values, change data fields) information. It is still based on a simple program design with a single instance object.

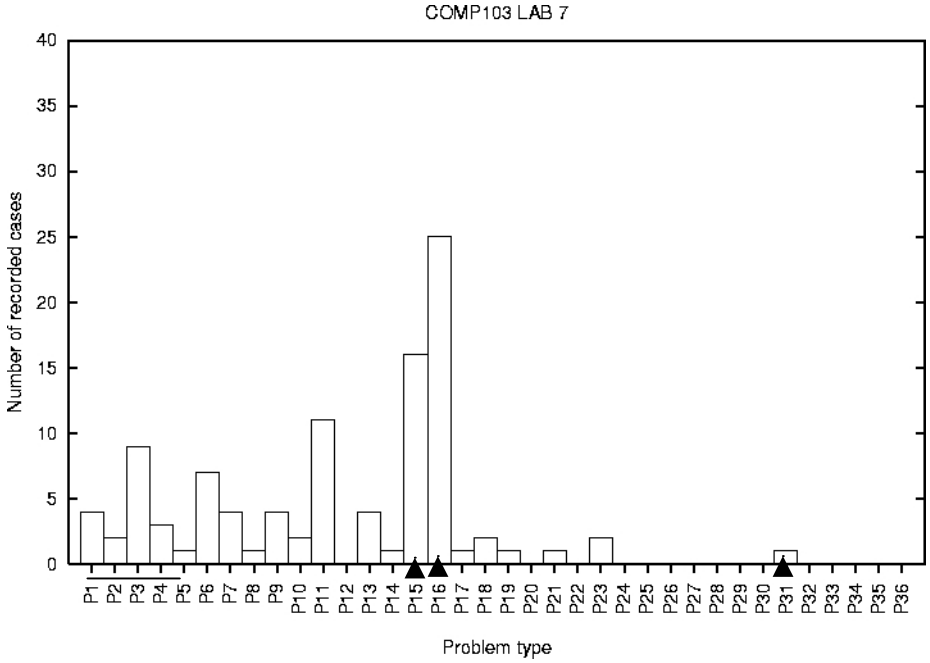
Task:

Input a name and age in days. Output a message involving a string manipulation and a calculation (age in years and days). Two methods with different ways of inputting and outputting information were required.

New language features:

The new language features introduced are parameters and return values, and simple string manipulations. The specific language related problem types (see Appendix A) expected are marked as described for Lab 4.

Problems observed:



Lab 9

This lab introduces inheritance and overriding for the first time. It is initially based on the same simple program design as earlier labs, with a single instance object. In the second part the support class is extended, and an instance of the child class is used.

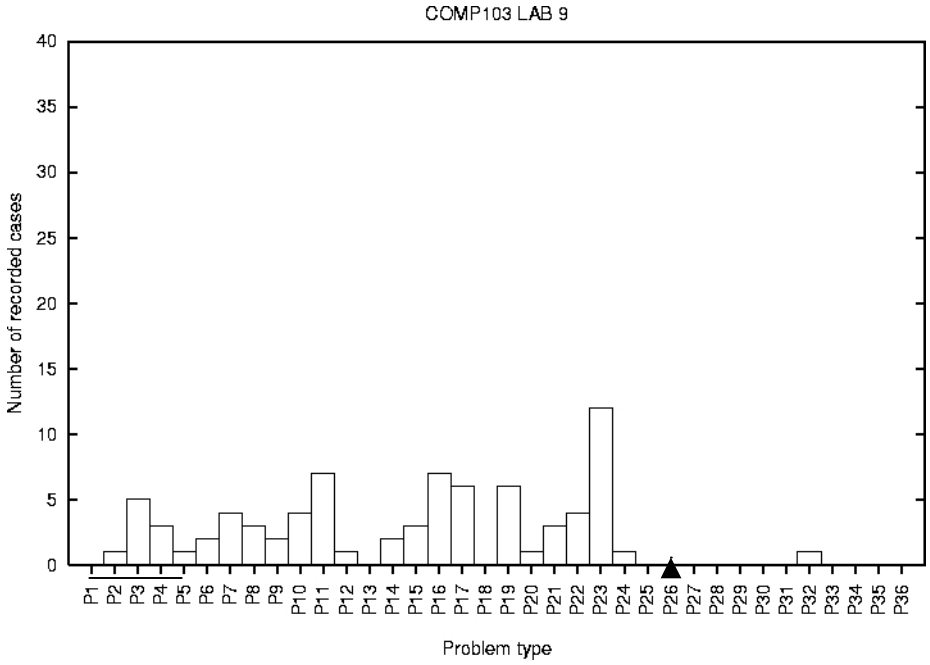
Task:

In the first part, input hours worked and hourly pay rate, output a message describing total pay. In the second part, extend the support class from the first part so that inputs include overtime hours and the total pay calculation is modified to include overtime.

New language features:

Inheritance and overriding are the only new features introduced. The language related problem types expected are marked as described for Lab 4.

Problems observed:



Lab 14

This lab introduces the first significant algorithmic complexity, involving loops and conditionals (if). Both topics had been met in previous labs, but in this lab they had to be combined in creative new ways.

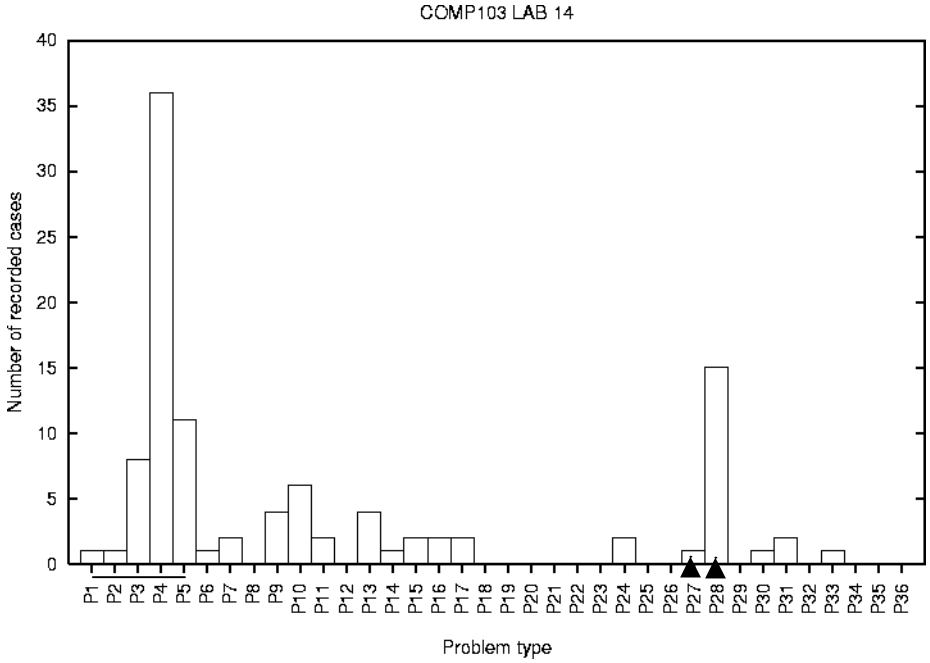
Task:

Given an input day (1 – 7) for the first of January, output a text calendar with one row per week, grouped by month, as for a regular calendar. Each month must start in the correct column.

New language features:

Although they had been met in previous labs we consider both loops and conditionals as new features. The language related problem types expected are marked as described for Lab 4.

Problems observed:



Lab 16

This lab tied together a number of topics which had been met in previous labs including files, arrays, multiple instance objects, constructors and accessors.

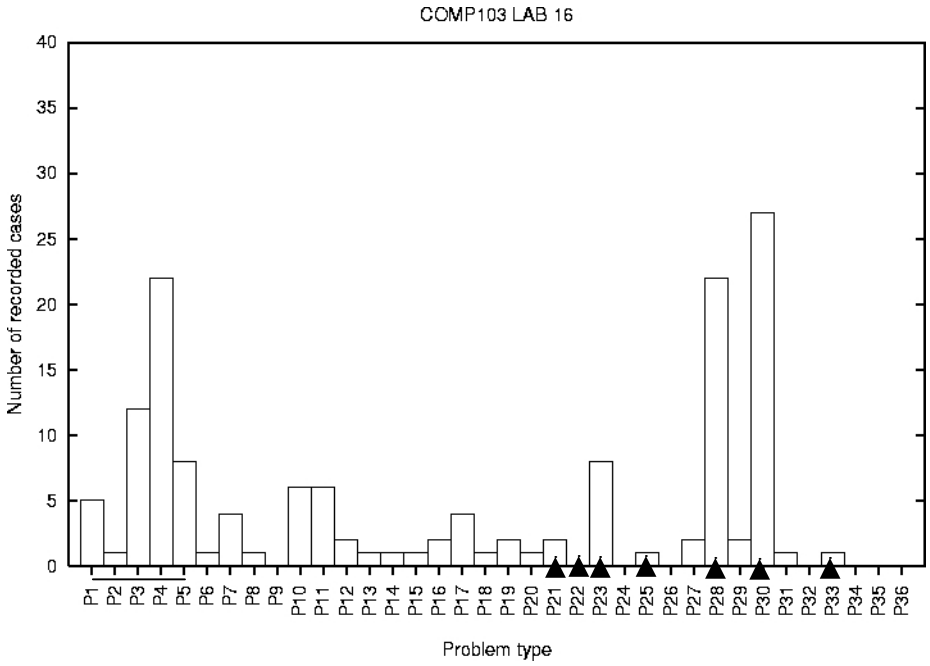
Task:

Design a class to hold information about students. Read data about individual students from a file, storing each record in a Student object in an array of Student objects. Process the records so that the individual and average grades are calculated.

New language features:

Although they had been met in previous labs we consider files, loops, arrays, and various OO concepts as new features. The language related problem types expected are marked as described for Lab 4.

Problems observed:



Lab 18

This lab is the second in a series of four, each building on the last to create a moderately complex and interesting program (which after the fourth lab involves multiple classes, inheritance, an array of objects, and some non–trivial algorithms). After these labs are completed the course moves on to applets and the lab work involves applets and simple GUIs.

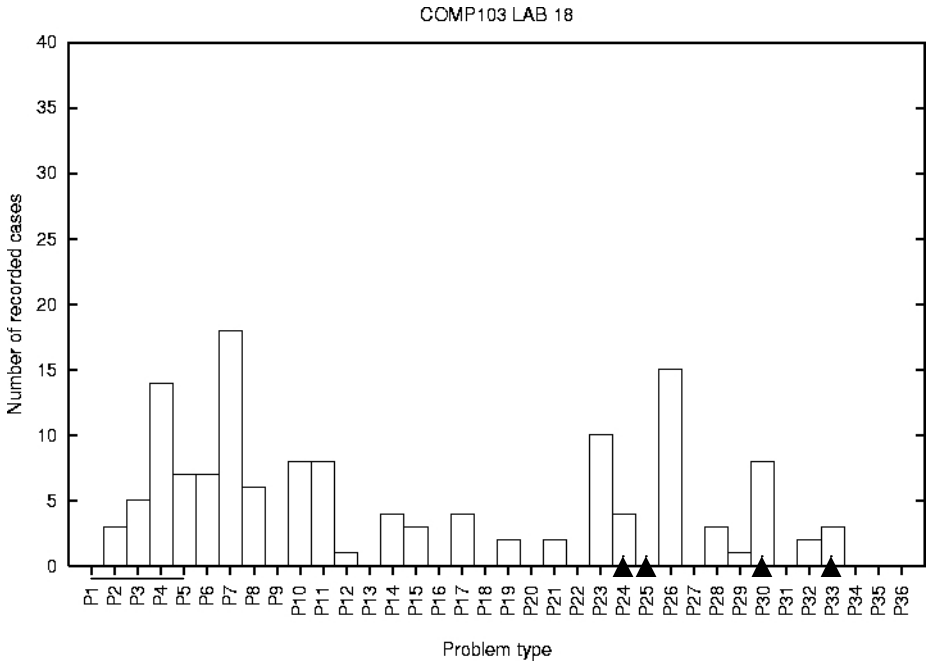
Task:

This task extends the program created in the previous lab. It involves creating a text based menu of options, reading in the user's choice, and displaying the selected information.

New language features:

The previous lab created a simple hierarchy of classes, thus reintroducing inheritance. This lab involved creating an array of objects as a static data field in the application class. (Although arrays of objects had been met in a previous lab we regard this as a new language feature here). This lab also introduced "raw" Java I/O (rather than the text book simpleIO package used so far). The language related problem types expected are marked as described for Lab 4.

Problems observed:



Lab 23

This lab is one of four at the end of the course which used applets (instead of applications) and developed simple GUIs.

Task:

Implement a calculator as an applet. The GUI is to consist of buttons for digits and simple functions (initially +, =, clear) and a label where the values entered and results are displayed.

New language features:

This lab involves elements of the Java AWT and event model, including buttons, an array of buttons, the ActionListener interface, an actionPerformed method and the use of the getSource method. Although some of these had been met in recent labs we regard them all as new features. The language related problem types expected are marked as described for Lab 4.

Problems observed: