The camel has two humps (working title)

Saeed Dehnadi and Richard Bornat
School of Computing, Middlesex University, UK

February 22, 2006

Abstract

Learning to program is notoriously difficult. A substantial minority of students fails in every introductory programming course in every UK university. Despite heroic academic effort, the proportion has increased rather than decreased over the years. Despite a great deal of research into teaching methods and student responses, we have no idea of the cause.

It has long been suspected that some people have a natural aptitude for programming, but until now there has been no psychological test which could detect it. Programming ability is not known to be correlated with age, with sex, or with educational attainment; nor has it been found to be correlated with any of the aptitudes measured in conventional 'intelligence' or 'problem-solving-ability' tests.

We have found a test for programming aptitude, of which we give details. We can predict success or failure *even before students have had any contact with any programming language* with very high accuracy, and by testing with the same instrument after a few weeks of exposure, with extreme accuracy. We present experimental evidence to support our claim. We point out that programming teaching is useless for those who are bound to fail and pointless for those who are certain to succeed.

1 Introduction

Despite the enormous changes which have taken place since electronic computing was invented in the 1950s, some things remain stubbornly the same. In particular, most people can't learn to program: between 30% and 60% of every university computer science department's intake fail the first programming course.[1] Experienced teachers are weary but never oblivious of this fact; bright-eyed beginners who believe that the old ones must have been doing it wrong learn the truth from bitter experience; and so it has been for almost two generations, ever since the subject began in the 1960s.

Computer scientists have long searched for a test for programming aptitude, if only to reduce the terrible waste of expectations that each first year represents, but so far to no avail. Programming aptitude does vary a little according to general educational attainment (we believe that we have evidence of this: see section 6), but it's far too weak an association to be of much use. John Lewis *are* well-advised to take their programmers from Oxbridge rather than polytechnics, there

[1] Nowadays in the UK one has to say that they *ought* to fail, but because of misguided Quality Assurance procedures and the efforts of colleagues who doggedly believe in the normal curve, very many of them are mistakenly and cruelly 'progressed' into following courses. That process so far degrades the quality of their education and the reputation of computer science as an academic discipline as to be of burning commercial, professional and intellectual importance, but in this paper it must be by the by.

are perhaps as many good programmers in high-class classics courses as in low-class computing courses, but the class prejudice nakedly obvious in those kinds of policies and judgements can't be allowed to stand. We really need an aptitude test.

It isn't just a UK problem. A study undertaken by nine institutions in six countries [22] looked at the programming skills of university computer science students at the end of their first year of study. All the participants were shocked by the results, which showed that at the conclusion of their introductory courses many students still did not know how to program. Lister and others, in [19], looked at seven countries and found the same thing. More recently, members of the same team [27] have looked for predictors of programming success and found only very weak indicators.

> "*Unfortunately, many students find it difficult to learn* [to program]. *Even universities that can select from among the best students report difficulties in teaching Java.*" [9]

Whatever the cause of the problem, it isn't a lack of motivation on anybody's part. Students joining programming courses are enthusiastic to find out how a set of instructions can apparently impart intelligence to an electronic machine, and teachers are keen to help them. But many find that they cannot learn what they want to know, however hard they try. They struggle on to the end of the course, for reasons of personal pride, family pressure or because of a lack of pastoral support and a dearth of escape routes, disillusioned and with a growing sense of personal failure. Those who can learn, on the other hand, are frustrated by the slow breeze of teaching, a wind tempered to the struggling shorn lambs. Teachers become demoralised that once again they've failed to crack the problem. Everybody tries their best, and almost everybody ends up unhappy – until the next year when a new intake begins a new course with their motivational batteries fully charged.

The cause isn't to be found in inappropriate teaching materials or methods either. Essentially, the computer science community has tried everything (see section 2) and nothing works. Graphics, artificial intelligence, logic programming languages, OOP, C, C++, PROLOG, Miranda: you name it, we've tried it. We've tried conventional teaching, lab-based learning by discovery and remedial classes. We've tried enthusiasm and cold-eyed logical clarity. *Nothing* makes a difference. Even the UK's attempt in the 1980s to teach the whole UK population to program on the BBC Micro ran into the sand.

Interviewing and testing the intake has no effect either. We've tried giving them tea, we've talked to their parents, we've asked if they have a computer in their bedroom (yes, but they only play games on it), we've tried giving them automatic acceptance on application. Some of us even tried accepting highly-motivated arts students (Tony Hoare in Belfast because he was once such a one, and one of us (Bornat) in London because he knew that A-level Maths wasn't a good predictor of success), only to find that the poor things failed at an even greater rate than their nerdy colleagues.

Snake oil is the industry's preferred remedy for programming failure (after class prejudice) and they apply it liberally to education. One of us recalls a meeting of the Conference of Professors and Heads of Computing at the University of East Anglia in the late 1990s, addressed by a snake-oil salesman who said that he could teach anybody to program in only fourteen days, and why were we making such a fuss? We listened politely but without conviction. As for class prejudice, the head of IBM UK once famously opined that he got better programmers by selecting classicists rather than computer programmers (he might have added, classicists who can row) and John Lewis, a major UK retailer, only recruits programmers from Oxbridge, does not take notice of field of study, and trains them itself from scratch.

Up to this point, then, the computer science community is essentially stuffed, up a creek without a paddle, in a box canyon looking at a blank wall, bereft of ideas. *Nil desperandum*: this paper shows

the way out, gives us a paddle, revitalises our enthusiasm and restores our flexibility by describing a predictive test for programming aptitude.

2 Related work

Our paper is the first step in a paradigm shift, so there isn't much related work. Most of the psychological research into programming learners has thrown little or no light; almost all the research into teaching methods has been vacuous and bedevilled by the Hawthorn effect; research on teaching materials is very similar; and research into programming languages and IDEs has got precisely nowhere.

There are some useful nuggets, though. Soloway was a pioneer in the area, counting, cataloguing and classifying novices' mistakes and misconceptions. Adelson and Soloway [1] reported that domain experience affects novices' ability to understand the sort of programming problems that they are asked to solve – essentially, domain knowledge helps. Bonar and Soloway [6] catalogued bugs, trying to use them to guess what their subjects were thinking as they programmed, and found that just a few types of bug cover almost all those that occur in novices' programs. In [5] the same authors put forward evidence to support the startling theory that prior knowledge of one programming language has a negative impact on novices' attempts to program in a second language. Soloway and Spohrer [28] looked at novices' backgrounds, and opined that skill in natural language seemed to have a great deal of impact on their conceptions and misconceptions of programming.

Putnam and others found that novices' misconceptions about the capabilities of computers could have a massive negative impact on their success at programming [26]. Many of their subjects tried to use meaningful names for their variables, apparently hoping that the machine would be able to read and understand those names and so perceive their intentions.

Benedict and du Boulay catalogued the difficulties that novices experienced [4]. They identified orientation (what programming is), machinery (what a computer is), notation, structure in formal statements, and pragmatics (how to debug). They also categorised students' mistakes into misapplications of analogies, overgeneralisations, and inappppropriate interactions of different parts of a program.

Pennington looked at the way that expert programmers understand problem domains and programs [24]. He found that, despite the admonition of the computer science establishment to construct programs top down, experts build them bottom-up. His experts knew no more about their programming language than novices did, but they did know a lot more about their problem domains, and they utilised that knowledge.

Perkins and others described novice learners' problem-solving strategies [25]. Some, called "stoppers", appeared to give up at the first difficulty. Others, called "movers" seemed to use natural language knowledge to get beyond an impasse.

Van Someren looked at novices learning Prolog [29]. He came to the conclusion that those who were successful had a mechanical understanding of the way that the language implementation – the Prolog virtual machine – worked. He also noticed that the difficult bits of Prolog – unification and depth-first search – gave rise to most difficulties.

Mayer expressed the opinion that the formal nature of programming (he called it syntactic knowledge) is the difficult bit [20]. He believed that experts were able to think semantically. In [21] he showed that people who know how their programs work do better than those who do not. Canas and others came to a similar conclusion [10].

Adamzadeh and others looked at debugging [2]. They found that the majority of good debuggers are good programmers, but not vice-versa.

Hewett tried to use the study of psychology to motivate learning to program [13]. His subjects seem to have enjoyed the experience, and no doubt learned some psychology and programming along the way.

Murnane related programming to psychological theories [23], specifically Chomsky's notion of natural language acquisition and Piaget's theories of education.

Programming teachers, being programmers and therefore formalists, are particularly prone to the 'deductive fallacy', the notion that there is a rational way in which knowledge can be laid out, through which students should be led step-by-step. One of us even wrote a book [8] which attempted to teach programming via formal reasoning. Expert programmers can justify their programs, he argued, so let's teach novices to do the same! The novices protested that they didn't know what counted as a justification, and Bornat was pushed further and further into formal reasoning. After seventeen years or so of futile effort, he was set free by a casual remark of Thomas Green's, who observed "people don't learn like that", introducing him to the notion of inductive, exploratory learning.

There is a vast quantity of literature describing different tools, or as they are known today Interactive Development Environments (IDEs). Programmers, who on the whole like to point and click, often expect that if you make programming point-and-click, then novices will find it easier. The entire field can be summarised as saying "no, they don't". It would be invidious to point the finger at any examples.

Lots of people have tried to invent a programming language that novices can understand. Some of the languages – LOGO and Miranda, for example – have survived on their merits, but none of the educational experiments produced any reproducible results. Once again, it would be invidious to pick out examples.

There has been some good work in theory. Thomas Green put forward the notion of *cognitive dimensions* to characterise programming languages and programming problems [12]. Like Detienne in [11], he rejects the notion that programming languages can be 'natural': they are necessarily artificial, formal and meaningless. He is able to measure the difficulty levels of different languages (some are much worse than others) and even of particular constructs in particular languages. If-then-else is good, for example, if you want to answer the question "what happened next?" but bad if the question is "why did *that* happen?", whereas Dijkstra's guarded commands are precisely vice-versa.

Johnson-Laird has contributed the idea of *mental models* to the study of people's competence in deductive reasoning. With Wason he discovered that people make systematic errors in reasoning [31], and that they tend to draw on the content of premises to inform their conclusion, in ways that aren't deductively valid [30]. The results seemed to contradict the notion that humans use formal rules of inference when reasoning deductively, but the authors couldn't put forward any alternative explanation.

Later, Johnson-Laird put forward the theory that individuals reason by carrying out three fundamental steps [14]:

1. They imagine a state of affairs in which the premises are true – i.e. they construct a mental model of them.

2. They formulate, if possible, an informative conclusion true in the model.

3. They check for an alternative model of the premises in which the putative conclusion is false. If there is no such model, then the conclusion is a valid inference from the premises.

Johnson-Laird and Steedman implemented the theory in a computer program that made deductions from singly-quantified assertions, and its predictions about the relative difficulty of such problems were strikingly confirmed: the greater the number of models that have to be constructed in order to draw the correct conclusion, the harder the task [18]. Johnson-Laird concluded that comprehension is a process of constructing a mental model [15], and set out his theory in an influential book [16]. Since then he has applied the idea to reasoning about Boolean circuitry [3] and to reasoning in modal logic [17].

We haven't, unfortunately, been able to find references to support the strongly-attested computer-science-folk wisdom that all attempts to correlate programming ability with any other kind of aptitude, and in particular with the results of 'intelligence' tests, have resoundingly failed. One of us was involved in an expensive but abortive attempt to devise an aptitude test in the 1990s, so the rumours are almost certainly true, but we would like to be able to cite research that has looked for and failed to find such correlations. Advice is sought.

3 The experiment

From experience it appears that there are three major semantic hurdles which trip up novice imperative programmers. In order they are:

- assignment and sequence;[2]
- recursion / iteration;
- concurrency.

Few programmers ever reach the concurrency hurdle, which is the highest of the three, and very high indeed. Recursion is conceptually difficult, and the proper treatment of iteration is mathematically complicated. Assignment and sequence, on the other hand, hardly look as if they should be hurdles at all: storage of / remembering information and doing one thing after another are part of everyday patterns of life and thought, and you might have expected (as at first do most teachers) that students' experience could be analogised into some kind of programming expertise. Not so: it is a real hurdle, and it comes at the very beginning of most programming courses. We decided to investigate it.

Our intention was to observe the mental models that students used when thinking about assignment instructions and short sequences of assignments. We did not use Johnson-Laird's theories of deductive reasoning directly, but we assumed that our subjects, in answering questions about a computer program, would necessarily use a mental model of the calculation described by the program. We hoped to be able to find out what those models are.

We expected that after a short period of instruction our novices would be fairly confused and so would display a wide range of mental models, but that as time went by the ones who were successfully learning to program would converge on a model that corresponds to the way that a Java program actually works. So we planned to administer a test just after the students had begun

[2] Novice declarative programmers' first hurdle appears to be argument substitution. Really the first hurdle is just getting started: 'orientation' was the name that Benedict and du Boulay gave it in [4].

1. Read the following statements and tick the correct answer in the front column.	The new values of a and b are:	

`int a = 10;`
`int b = 20;`
 	
`a = b;`
	☐ `a = 30` `b = 0`
	☐ `a = 30` `b = 20`
	☐ `a = 20` `b = 0`
	☐ `a = 20` `b = 20`
	☐ `a = 10` `b = 10`
	☐ `a = 10` `b = 20`
	☐ `a = 20` `b = 10`
	☐ `a = 0` `b = 10`
	☐ **If none, give the correct values:**
	a = **b =**	

Figure 1: A sample test question

to be taught about assignment and sequence, at the very beginning of their course, then a second time to the same subjects after the topic had been taught, and then a third time just before the examination. We planned to correlate the results of these three administrations with each other and also with the subjects' marks in the official end-of-course examination.

Dehnadi devised a test, with questions as illustrated in figure 1. Each question gave a sample Java program, declaring two or three variables and executing one, two or three variable-to-variable assignment instructions. We had some prior notion, from our teaching experience, of the ways that a novice might understand the programs, and Dehnadi invented mental models accordingly. In this he was assisted by the decision of the Java designers to use the equality sign '=' to indicate assignment.[3] The first column, apart from the word 'int' and the three semicolons, contains formulae which we expected that our subjects would have encountered in their earlier mathematical studies.

The middle column gives a multiple-choice list of the alternative answers given by Dehnadi's collection of mental models. By accident he missed out two models which, by symmetry, he ought to have included – so in figure 1, for example, there should have been ten elements in the list, rather than eight – but in the event it didn't cause us any difficulties. In the middle column '=' means equality: this doesn't seem to have confused any of our subjects. Note the word 'new' at the head of the column: this seems to have given most of them a hint that *change* of values is at the heart of the question, whatever they may have thought about algebraic equality beforehand.

The blank third column was for rough work: we found the marks that the subjects made here occasionally illuminating.

The mental models of assignment that we expected our subjects to use are shown in table 1. Note that model 10 (equality) overlaps with models 2 (right-to-left copy) and 4 (left-to-right copy): the difference could be detected, if necessary, by looking at questions which included more than a single assignment instruction.

3.1 First administration

In a bizarre event which one of the authors insists was planned, and the other maintains was a really stupid idea that just happened to work, the test was first administered to about 30 students on a

[3] As computer scientists we normally excoriate this appalling design decision, but here we are able to benefit from their crass idiocy.

Table 1: Anticipated mental models of assignment

1. Value moves from right to left (a := b; b := 0 – third line in figure 1).
2. Value copied from right to left (a := b – fourth line of figure 1, and the 'correct' answer).
3. Value moves from left to right (b := a; a := 0 – eighth line of figure 1).
4. Value copied from left to right (b := a – fifth line of figure 1, and a reversed version of the 'correct' answer).
5. Right-hand value added to left (a := a+b – second line of figure 1).
6. Right-hand value extracted and added to left (a := a+b; b := 0 – first line of figure 1).
7. Left-hand value added to right (b := a+b – omitted in error).
8. Left-hand value extracted and added to right (b := a+b; a:=0 – omitted in error).
9. Nothing happens (sixth line of figure 1).
10. A test of equality: nothing happens (fourth and fifth lines of figure 1).
11. Variables swap values (seventh line in figure 1).

further-education programming course at Barnet College *before they had received any programming teaching whatsoever* – that is, in week 0 of the course. Those students had no particular pattern of age, sex and educational background. Dehnahdi interviewed half of them before admission, and taught them all. We believe that none had any previous contact with programming, and that all had enough school mathematics to make the equality sign familiar. The test was anonymised, in that the students invented nicknames for themselves.

The same test was then administered to about 30 students in the first-year programming course at Middlesex University, once again before they had received any programming teaching. They were mostly male, aged about 18-20, from the middle range of educational attainment and from families towards the lower end of income distribution. Again the answers were anonymised. This time Dehnahdi tutored them but did not teach the course. We believe (without any interview evidence at all) that they had had no contact with programming but had received basic school mathematics teaching.

There was an attempt to administer the test to another 30 or so students on a foundation (pre-degree) programming course at Middlesex University. That administration failed, because the students – rightly, in our opinion – were incensed at the conduct of their teaching and the arrangements for their study, and simply refused to do anything that wasn't directly beneficial to themselves.

In total, the test was successfully administered to 61 subjects.

3.2 Second administration

The test was administered again to the same subjects (with the exception of the rebellious foundation year) in week 3. Anonymisation introduced difficulties, because some students had forgotten

For a non-trivial programming problem, which one of the following is an appropriate language for expressing the initial stages of algorithm refinement?

(a) A high-level programming language.

(b) English.

(c) Byte code.

(d) The native machine code for the processor on which the program will run.

(e) Structured English (pseudocode).

Figure 2: A bookwork multiple-choice question from the week 7 in-course exam (1 mark)

Consider the following program fragment.

```
if (mark > 80) grade = 'A';
else if (mark > 60) grade = 'B';
else if (mark > 40) grade = 'C';
else grade = 'F';
```

What is the effect of executing this code if the variable `mark` has the value -12?

(a) The program will crash.

(b) The program will output an error message.

(c) The variable `grade` will be undefined.

(d) The program will never terminate.

(e) The variable `grade` will be assigned the value 'F'.

Figure 3: A technical multiple-choice question from the week 7 in-course exam (1 mark)

their nicknames. Comparison of handwriting and some inspired guesses mean that we got them all.

3.3 Third administration

Because of what we found on the first and second administrations, there was no third administration.

3.4 Examinations

Students had to undertake a summative end-of-course examination in week 12. There were also two in-course formative examinations taken in weeks 7 and 11 at Middlesex, and at Barnet they took the same in-course exam in weeks 6 and 10. The week 7 exam contained sixteen multiple-choice questions ranging from trivial bookwork (e.g. figure 2) to technical analysis (e.g. figure 3), and two creative questions (e.g. figure 4). The week 11 exam had write-in bookwork questions (e.g. figure

The Hendon Used Car dealership price list shows the price of used cars with a mileage of exactly 10000 miles, and adjusts the prices of its cars depending on their individual mileage, as shown in this table:

Price adjuster for mileage	
Under 10,000 miles	add 200
Over 10,000 miles	subtract 300

Assuming that the price and mileage of a given car are stored in integer variables `price` and `miles`, write `if..else` statement(s) that adjust the value of `price` according to the table above.

Figure 4: A technical creative question from the week 7 in-course exam (5 marks)

The _____ -controlled loop is used to do a task a known number of times.

Figure 5: A bookwork write-in question from the week 11 in-course exam (1 mark)

What is wrong with the following main method which is intended to add the numbers from 1 to 4 together and print out the result which should be 10? Give a correct version:

```
public static void main (String[] args) {
   int total = 0;
   for (int i = 1; i < 4; i++)
      total = total+i;
   System.out.println("Total:  " + total);
}
```

Figure 6: A technical write-in question from the week 11 in-course exam (1 mark)

The following code is an incomplete version of a 'guess the number' program, in which the user is repeatedly invited to guess a hidden number. Add the necessary code to the body of the while loop to check the user's guess and output appropriate messages depending on whether or not the guess is correct.

```
final int HIDDENNUM = 20;

String numStr =
   JOptionPane.showInputDialog("Enter your guess (zero to finish)");
int guess = Integer.parseInt(numStr);

while (guess != 0) {

   //ADD YOUR CODE HERE

   String numStr =
     JOptionPane.showInputDialog("Enter your guess (zero to finish)");
   int guess = Integer.parseInt(numStr);
}
```

Figure 7: A technical creative question from the week 11 in-course exam (5 marks)

5), technical write-in questions (e.g. figure 6) and longer creative questions (e.g. figure 7)

3.5 Correlation

We correlated the results of the first and second administrations, using the code of anonymity. With our subjects' assistance we broke the code and correlated the test resul;ts with the in-course exam results and also with the pass/fail results of the end-of-course exam (the only publicly-available information).

4 Results

It seems absurd to administer a test to subjects who can have no idea what the questions mean. Humans are clever animals, though, and most of our subjects seem to have guessed, perhaps from the use of 'new' in the heading of the multiple choice column, that the questions were about change rather than mathematical equality, and so they answered using a variety of models. There were almost no write-in answers.

4.1 Three groups

We could hardly expect that students would choose the Java model of assignment (model 2 in table 1), but it rapidly became clear that despite their various choices of model, in the first administration they divided into three distinct groups with no overlap at all:

- 44% used the same model for all, or almost all, of the questions. We call this the ***consistent*** group.
- 39% used different models for different questions. We call this the the ***inconsistent*** group.
- The remaining 8% refused to answer all or almost all of the questions. We call this the ***blank*** group.

We did not interview our subjects to determine anything about their group membership, so we do not know whether students chose consciously or unconsciously to follow one strategy or another, nor how conscious choices (if any) were motivated, nor what any particular choice meant to a subject who made it. We have no information about how group membership correlates with earlier education, employment experience, age, sex, marital status or indeed anything else.

Speculation is part of science, though, and those to whom we have recounted this tale have usually been ready to try it. Told that there were three groups and how they were distinguished, but not told their relative sizes, we have found that computer scientists and other programmers have almost all predicted that the blank group would be more successful in the course exam than the others: "they had the sense to refuse to answer questions which they couldn't understand" is a typical explanation. Non-programming social scientists, mathematicians and historians, given the same information, almost all pick the inconsistent group: "they show intelligence by picking methods to suit the problem" is the sort of thing they say. Very few, so far, have predicted that the consistent group would be the most successful. Remarkably, it is the consistent group, and almost exclusively the consistent group, that is successful. We speculate on the reasons in section 7.

Table 2: Stable group membership between first and second administrations

	Consistent (A2)	Inconsistent (A2)	Total
Consistent (A1)	25	2	27
Inconsistent (A1)	11	13	24
Blank (A1)	5	5	10

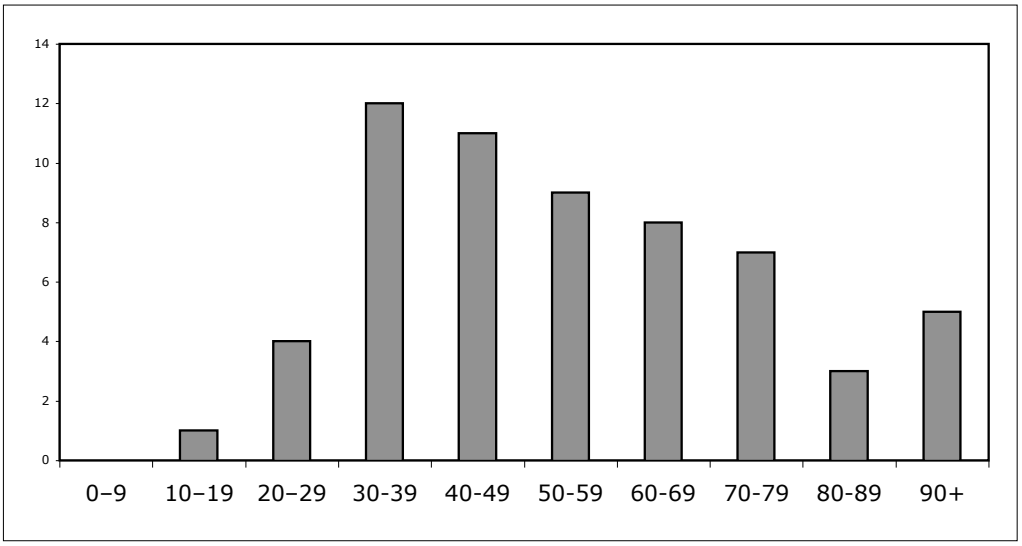

Figure 8: Average in-course exam results

4.2 Correlation of the two test administrations

Table 2 shows that there is very little movement from consistent to inconsistent between the two test administrations: the two people who did switch appear not to have attended the course at all, and may not have taken the second administration seriously. There is some movement – almost half – from inconsistent or blank to consistent. Everyone became either consistent or inconsistent: there were no blank returns in the second test.

4.3 Correlation with exam marks

The average of the two in-course exam results, one from week 7 and the other from week 11, are plotted in figure 8: hardly a classic double-hump curve, but by no means a normal curve either. When redrawn in figure 9 to distinguish those who had been consistent in the first test (black) from those who had been inconsistent or blank (white) it is clear that we do have at least two humps – a white one at 40 and a black one at 60 – as well as a hint of a second white hump centred on 75 and a second black hump in the 90+ column.

Table 3 shows the numbers of passes and fails in the end-of-course examination for the consistent group in the first test and the rest. Despite the stark difference in success between the groups, there are only two data points per column. In the rest of the discussion we shall therefore consider only the in-course exam results.

The switchers from inconsistent/blank to consistent in table 2 are almost indistinguishable from

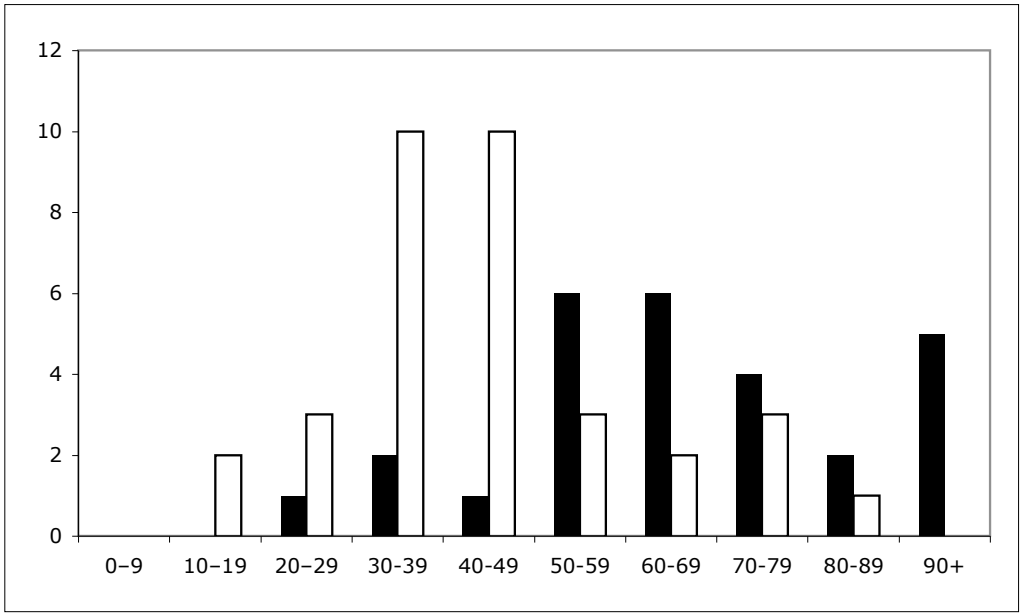

Figure 9: First-test consistent (black) and inconsistent/blank (white) average in-course exam results

Table 3: End-of-course exam pass/fail statistics, correlated against first-test result

	Consistent(A1)	Inconsistent/Blank (A1)
Pass	21	8
Fail	6	26

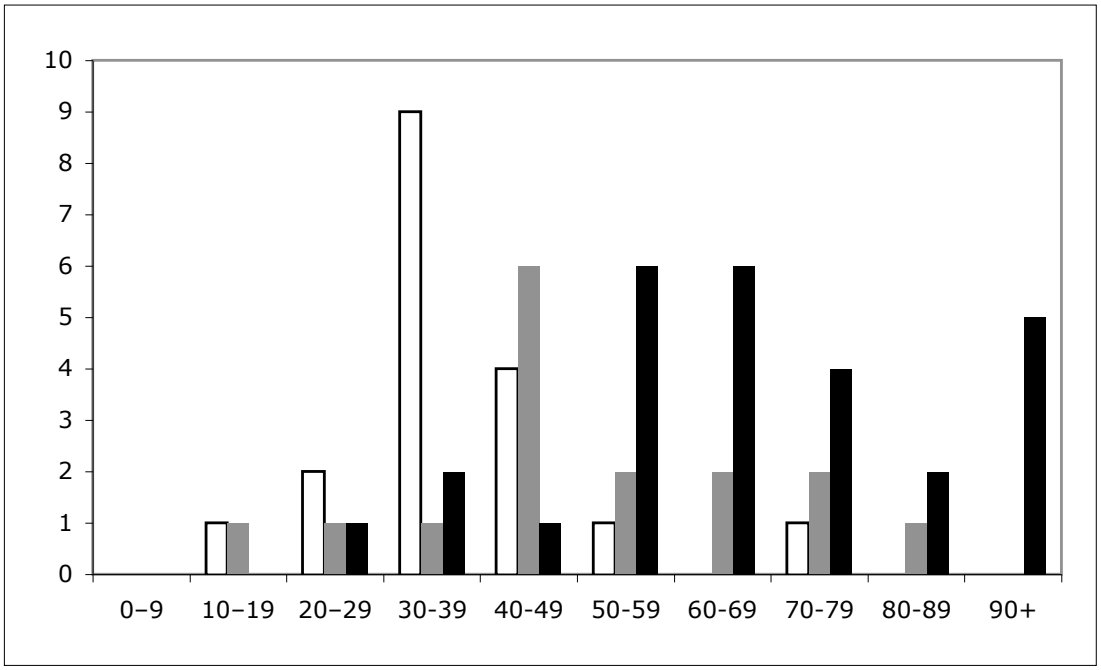

Figure 10: Second-test inconsistent (white), second-test consistent switchers (grey) and first-test consistent (black) average in-course exam results

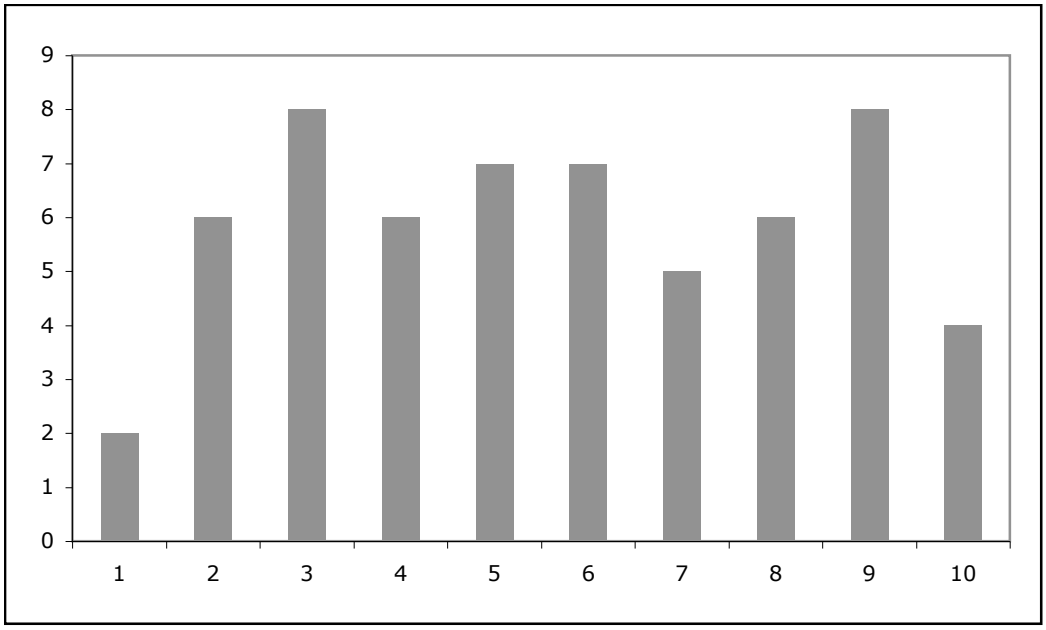

Figure 11: Marks in the week 11 in-course exam

those who stay put. Figure 10 compares the three populations. The switchers (grey) are a little bit farther to the right – that is, get slightly higher marks – than the non-switchers (white), but they are clearly distinguishable from the consistent group (black). The only noticeable difference between figures 9 and 10 is that the non-switchers form the left half of the 30-49 hump and the switchers form the right half, a considerably underwhelming result.

4.4 Correlation with second in-course exam

We correlated our test with the average of the week 7 and week 11 exams because that is what Middlesex University does with the marks, and we wanted to see how our test stood up in practice. But the week 7 exam contains a lot of multiple-choice questions and doesn't give negative marks for wrong answers: in those circumstances guessers can score highly. The week 11 exam also comes later and tests the whole range of the course. For both those reasons, it is likely to be a much better examination of programming ability.

Figure 11 shows the distribution of marks in the week 11 in-course exam, and there is obviously more than one hump. When the consistent/not-consistent groups are separated in figure 12 the difference is plain. There seem even more clearly to be two sub-populations of consistency, with one black hump centred at 60 and another at 85. There also seems to be evidence, in the white hump centred at 80, that some of the inconsistent/blank group can learn to program.

Figure 13 looks at the week 11 marks of the same three populations as figure 10. There is evidence in the 70-79 and 80-89 columns that some switchers *can* program, and in the 70-79 column it seems that one of the non-switchers can program too. The first-test consistent group, though overwhelmingly successful in passing the exam (scoring 40 or more) can clearly be seen to be made up of two sub-populations. One, with a hump at 85, is extremely successful. The other, with a hump at 60, is moderately successful. The second-test inconsistent group, on the other hand, can be almost entirely written off.

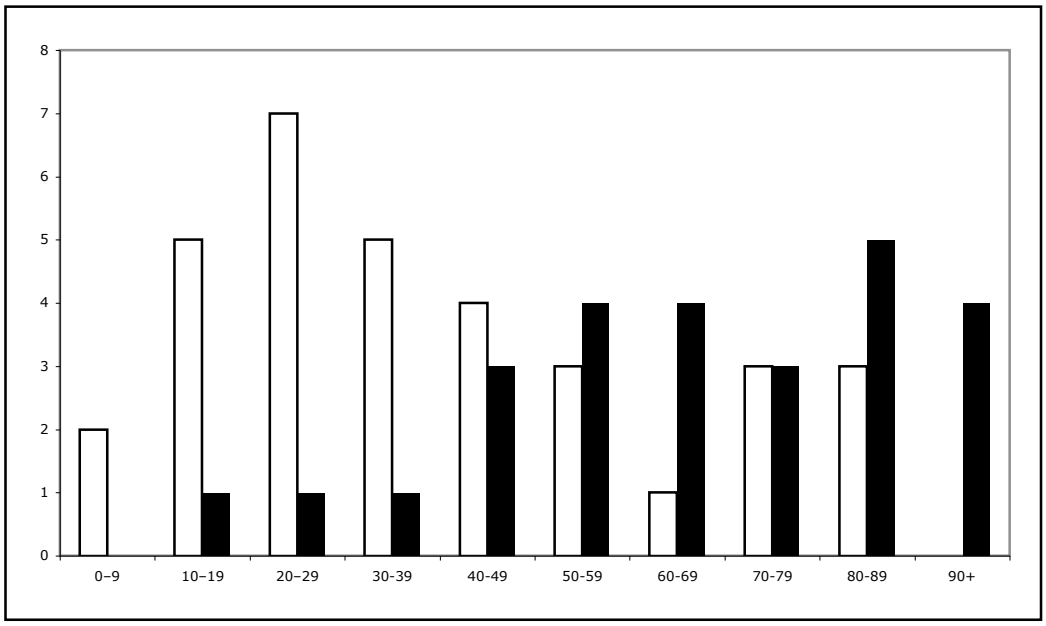

Figure 12: First-test consistent (black) and inconsistent/blank (white) week 11 exam results

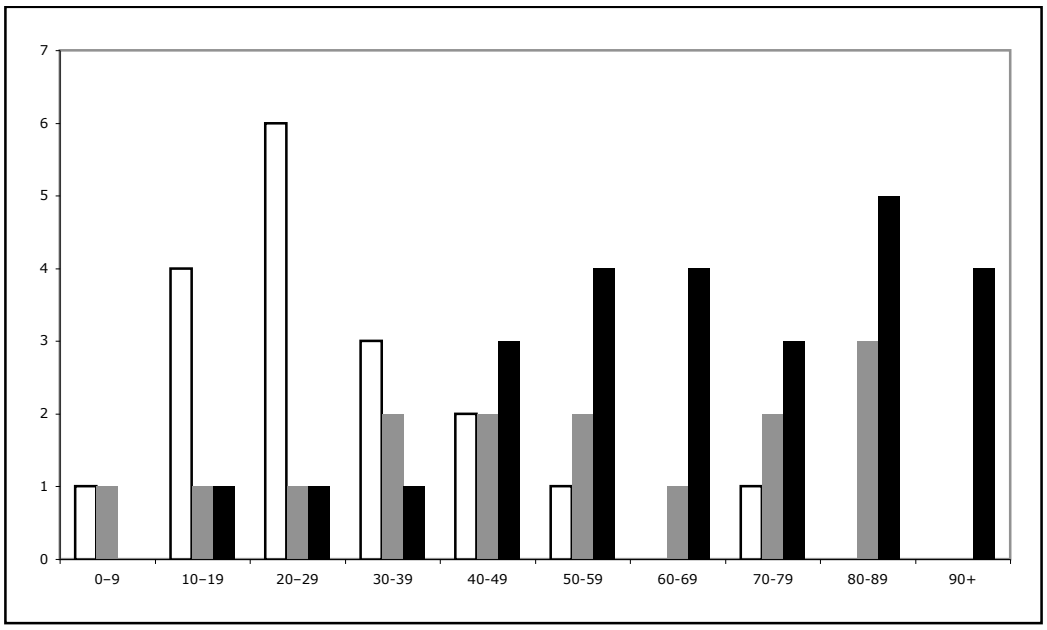

Figure 13: Second-test inconsistent (white), second-test consistent switchers (grey) and first-test consistent (black) week 11 exam results

5 Analysis and speculation

Every experienced programming teacher believes that there at least two populations in initial programming courses. Figure 9 suggests that there are two or perhaps three, and figure 10 doesn't undermine that impression. At QMC between 1975 and 1990 we used to believe that there were three populations: one which did very well, another which managed to get by, and a third which had no idea what on earth programming was about. Figure 13 seems to support that view.

Figure 9 shows that the first administration of Dehnadi's test reliably separated the consistent group, who almost all scored 50 or above, from the rest, who almost all scored below 50, with only 4 out of 27 false positives in the consistent group and 9 out of 34 false negatives in the rest. Table 3 tells almost exactly the same story: hardly surprising, since the final results contained a considerable component taken from the in-course examinations. Clearly, Dehnahdi's test is not a perfect divider of programming sheep from non-programming goats. Nevertheless, if it were used as an admissions barrier, and only those who scored consistently were admitted, the pass/fail statistics would be transformed. In the total population 32 out of 61 (52%) failed; in the first-test consistent group only 6 out of 27 (22%). We believe that we can claim that we have a *predictive* test which can be taken prior to the course to determine, with a very high degree of accuracy, which students will be successful. This is, so far as we are aware, the first test to be able to claim any degree of predictive success.

There is evidence in figure 13, although we must be cautious given the small numbers involved, that a small proportion of the first-test inconsistent/blank group respond to teaching and become good programmers by week 11 of the course.

It has taken us some time to dare to believe in our own results. It now seems to us, although we are aware that at this point we do not have sufficient data, and so it must remain a speculation, that what distinguishes the three groups in the first test is their different attitudes to meaninglessness.

Formal logical proofs, and therefore programs – formal logical proofs that particular computations are possible, expressed in a formal system called a programming language – are *utterly meaningless*. To write a computer program you have to come to terms with this, to accept that whatever you might want the program to mean, the machine will blindly follow its meaningless rules and come to some meaningless conclusion. In the test the consistent group showed a pre-acceptance of this fact: they are capable of seeing mathematical calculation problems in terms of rules, and can follow those rules wheresoever they may lead. The inconsistent group, on the other hand, looks for meaning where it is not. The blank group knows that it is looking at meaninglessness, and refuses to deal with it.

Figure 13 suggests that it is extremely difficult to teach programming to the inconsistent and blank groups. It *might* be possible to teach them, if we concentrated on trying to persuade them to see a programming language as a system of rules (though the evidence in section 6 below seems to suggest otherwise).

The consistent group seem to be much easier to teach. We speculate that they divide into two groups: the very successful ones find programming easy and may perhaps be those who ought to follow a conventional computer-science education, with lots of formal content; the moderately successful perhaps are the software engineers, those who can program but can't imagine that they will ever enjoy it, and are content to submit to management discipline and to be drowned in UML (ugh!).

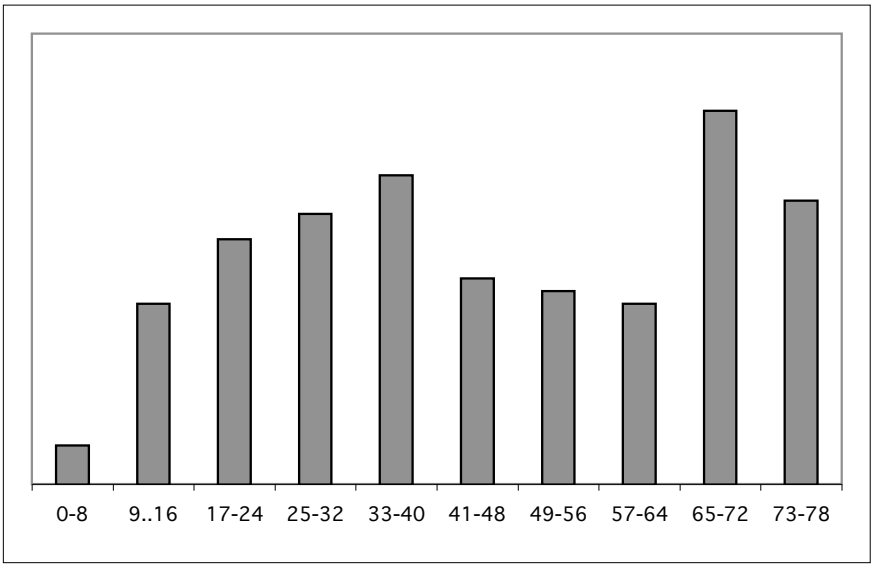

Figure 14: Double-humped exam results in formal proof

6 Teaching formal logic

One of us (Bornat), disillusioned and dispirited after 30 years of trying to teach programming and 30 years of failure to eradicate the double hump, turned to formal logic. He had developed the proof calculator Jape [7], and hoped that with its assistance students could learn to make formal proofs. He speculated that the size of programming languages might confuse many students trying to learn to program. Java, the teacher's programming language of choice at the end of the 20th century and at the beginning of the 21st, is defined loosely but at great length in several hundred pages.[4] Natural Deduction, on the other hand, can be defined precisely on a single side of a single A4 sheet, in a normal font size. It is small and conceptually simple: surely it can be taught!

Well, up to a point it can, but the results were depressingly familiar. Figure 14 shows the results from an in-course exam on formal logical proof. Unlike the examinations of section 3, students were strongly discouraged from guessing, with negative marks for wrong answers in multiple-choice questions and extensive unseen proof problems. As a consequence, two humps are clearly visible, with one at 33-40 (about 50%) and another at 65-72 (about 85%). (This data was collected long before Dehnadi's test was developed, so we can't analyse it further).

There is more. Within the course there were several sub-groups, due to the vagaries of university admissions procedures. One group was admitted to a computer science course and had higher than average A-level scores within the population of students admitted to UK universities: we call this group the **high achievers**. Another group had very much lower than average A-level scores, and had been admitted to a maths and computing course largely to boost numbers in the mathematics department: we call this group the **low achievers**. When we separate out the formal proof test scores of the two groups, in figure 15, we can see that the high achievers have a large success hump centered on 72 – about 90% – and a somewhat smaller failure hump centred on 33-40 – about 50%. The low achievers have a much smaller success hump, also centred on 72, and a much larger failure

[4] Most of that definition is of the library: you need to know about a considerable part of the library before you can do very much in Java, so you have to engage with it almost from the start.

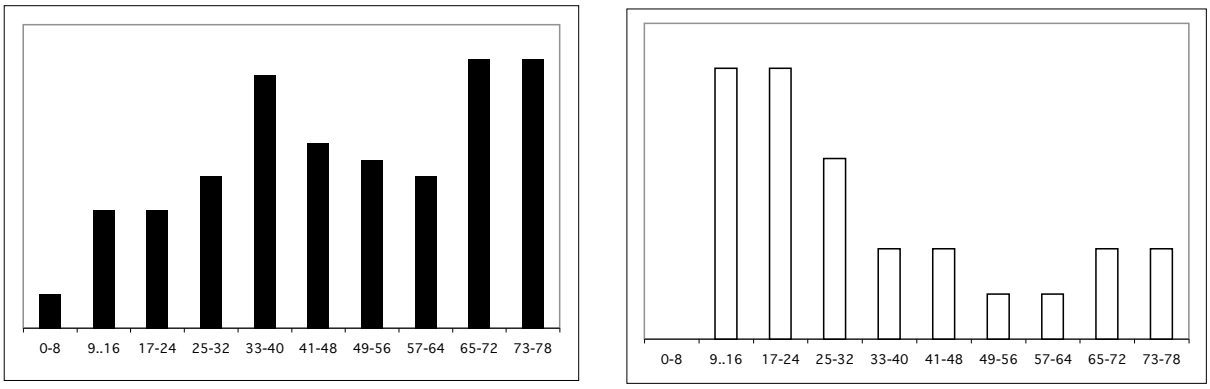

Figure 15: High achievers' (black) and low achievers' (white) exam results in formal proof

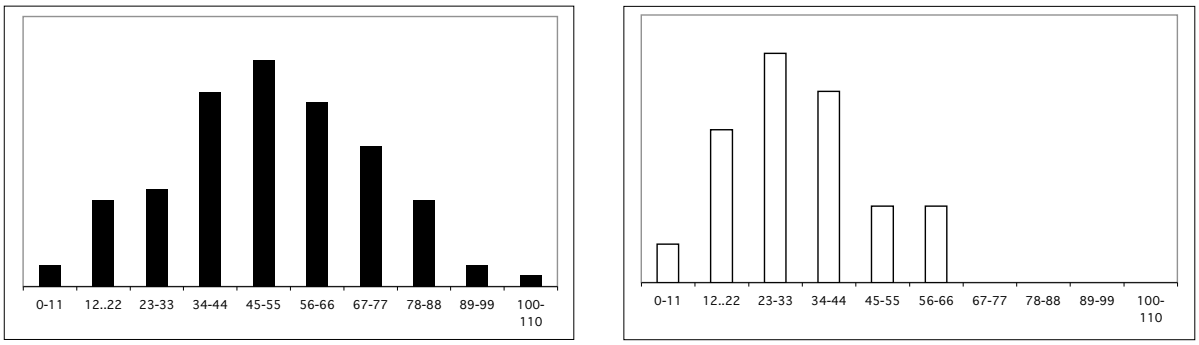

Figure 16: High achievers' (black) and low achievers' (white) exam results in informal disproof

hump centred on 16 – about 20%.

We speculate from this data that the proportion of formal provers / programmers in the general population varies according to general educational attainment. It would seem, from our Middlesex data, to be less than 50%; we speculate that it is about 40%.

There is still more. The *same students*, on the *same course*, when given an informal topic – constructing Kripke trees to demonstrate counter-examples to constructive logical assertions, a task which involves assigning meaning to formulae and is therefore not at all like programming – generated normal curves in a test. Figure 16 shows the data. The high achievers' hump is well within the passing zone, if a little bit lower than the administrator-desired 55%, whereas the low achievers' hump is well below the pass mark of 40%. But a normal curve for all that!

7 Conclusion

There is a test for programming aptitude, or at least for success in a first programming course. We have speculated on the reasons for its success, but in truth we don't understand how it works any more than you do. An enormous space of new problems has opened up before us all.

Acknowledgements

We are grateful to Middlesex University and to Barnet College for giving us permission to test their students, and to them and to the subjects themselves for allowing us access to their in-course examination results. We thank Ed Currie for permission to reproduce the material in figures 2 to 7.

References

[1] B Adelson and E Soloway. The role of domain experience in software design. *IEEE Transactions on Software Engineering*, 11(November):1351–1360, 1985.

[2] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 84–88. ACM Press, 2005.

[3] M.I. Bauer and P.N. Johnson-Laird. How diagrams can improve reasoning: Mental models and the difficult cases of disjunction and negation. In *Proceedings of the Annual Conference of Cognitive Science*, Boulder, Colorado, 1993. Lawrence Erlbaum Associates.

[4] J H Benedict and B Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[5] J Bonar and E Soloway. Pre-programming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*, 1(2):133–161, 1985.

[6] Jeffrey Bonar and Elliot Soloway. Uncovering principles of novice programming. In *10th ACM POPL*, pages 10–13, 1983.

[7] Richard Bornat. Jape software. `http://www.jape.org.uk`.

[8] Richard Bornat. *Programming from First Principles*. Prentice/Hall International, 1986.

[9] Tom Boyle, Claire Bradley, Peter Chalk, Ken Fisher, Ray Jones, and Poppy Pickard. Improving pass rates in introductory programming. In *4th Annual LTSN-ICS Conference*, NUI Galway, 2003.

[10] J. J Canas, M. T Bajo, and P Gonzalvo. Mental models and computer programming. *Journal of Human-Computer Studies*, 40(5):795–811, 1994.

[11] F. Detienne. Difficulties in designing with an object-oriented language: An empirical study. In D. Gilmore G. Cockton D. Diaper and B. Shackel, editors, *Human-Computer Interaction*, volume 3rd IFIP INTERACT 90, pages 971–976. North-Holland, Cambridge, 1990.

[12] Thomas Green. Cognitive approach to software comprehension: Results, gaps and introduction. In *Workshop on Experimental Psychology in Software Comprehension Studies*, University of Limerick, 1997.

[13] Thomas T. Hewett. An undergraduate course in software psychology. *SIGCHI Bull.*, 18(3):43–49, 1987.

[14] P.N. Johnnson-Laird. Models of deduction. In R Falmagne, editor, *Reasoning: Representation and Process*. Erlbaum, Springdale, NJ, 1975.

[15] P.N. Johnson-Laird. Comprehension as the construction of mental models. *Philosophical Transactions of the Royal Society, Series B*, 295:353–374, 1981.

[16] P.N. Johnson-Laird. *Mental Models*. Cambridge University Press, Cambridge, 1983.

[17] P.N. Johnson-Laird and V.A. Bell. A model theory of modal reasoning. In *Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society*, pages 349–353, 1997.

[18] P.N Johnson-Laird and M.J. Steedman. The psychology of syllogisms. *Cognitive Psychology*, 10:64–99, 1978.

[19] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Mostr, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers, 2004.

[20] Richard E Mayer. The psychology of how novices learn computer programming. In E. Spohre and J.C. Soloway, editors, *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, 1989.

[21] Richard E Mayer. *Thinking, Problem Solving, Cognition*. W. H. Freeman and Company Second Edition ISBN 0716722151, New York, 2 edition, 1992.

[22] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, Canterbury, UK, 2001. ACM Press.

[23] J Murnane. The psychology of computer languages for introductory programming courses. *New Ideas in Psychology*, 11(2):213–228, 1993.

[24] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19:295–341, 1987.

[25] D.N Perkins, C Hancock, R Hobbs, F Martin, and R Simmons. Conditions of learning in novice programmers. In E. Soloway Spohrer and J. C., editors, *Studying the Novice Programmer*, pages 261–279. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[26] Ralph T Putnam, D Sleeman, Juliet A Baxter, and Laiani K Kuspa. A summary of misconceptions of high school basic programmers. *Journal of Educational Computing Research*, 2(4), 1986.

[27] Simon, S. Fincher, A. Robins, B. Baker, I. Box, Q. Cutts, M. de Raadt, P. Haden, J. Hamer, M. Hamilton, R. Lister, M. Petre, K. Sutton, D. Tolhurst, and J. Tutty. Predictors of success in a first programming course. In *Proc. Eighth Australasian Computing Education Conference (ACE2006)*, pages 189–196. ACS, 2006.

[28] E Soloway and James C Spohrer. Some difficulties of learning to program. In E Soloway and James C Spohrer, editors, *Studying the Novice Programmer*, pages 283–299. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[29] Maarten W. van Someren. What's wrong? Understanding beginners' problems with Prolog. *Instructional Science*, 19(4/5):256–282, 1990.

[30] P. C. Wason and P.N. Johnson-Laird. *Psychology of Reasoning: Structure and Content*. Batsford, London, 1972.

[31] P.C. Wason and P.N. Johnson-Laird. *Thinking and Reasoning*. Harmondsworth: Penguin, 1968.