

The camel has two humps (working title)

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

January 10, 2006

Abstract

Learning to program is notoriously difficult. A substantial minority of students fails in every introductory programming course in every UK university. Despite great academic effort, the proportion has increased rather than decreased over the years, and 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. It is not correlated with age, with gender, or with educational attainment; nor is it 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. Remarkably, we can predict success or failure *even before students have had any contact with any programming language*, and with *total accuracy*. We present statistical analyses to prove the latter point. We speculate that programming teaching is therefore ineffective for those who are bound to fail and pointless for those who are bound to succeed.

1 Introduction

Despite the enormous changes which have taken place since electronic computing was invented in the 1950s, some things in the industry remain stubbornly the same. In particular, most people can’t learn to program: typically between 30% and 60% of a university computer science department’s intake fail the first programming course.¹ Experienced teachers are weary of the fact. Bright-eyed novice teachers who begin by believing that everybody must have been doing it wrong gain bitter experience, and so life goes on.

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* right to take their programmers from Oxbridge rather than polytechnics, there *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.

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 computer science education as to be of burning academic, professional and intellectual importance, but in this paper it must be by the by.

It isn't just a UK problem. A study undertaken by nine institutions in six countries [31] 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 [28], looked at seven countries and found the same thing.

“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.” [10]

Whatever is 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. Many of them are disappointed to find that they cannot learn how to do it. For family reasons or because of a lack of pastoral support and a dearth of escape routes (in universities outside Oxbridge), they generally struggle on to the end of the course, disillusioned and with a sense of personal failure. Those who can learn, on the other hand, are frustrated by the slow pace of the teaching, which is moderated to the perceived needs of the strugglers. 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 everybody starts again with their motivational batteries fully charged.

The cause isn't 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, remedial classes, everything. 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 with the BBC Micro came to nothing.

Interviewing and testing the intake has no effect either. We've tried giving them tea, we've talked to their parents, we ask them 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 racism) 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, and I left by bus shortly afterwards, in despair. 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 from Oxbridge and does not take notice of field of study.

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*, because this paper shows the way out, gives us a paddle, revitalises our enthusiasm, restores our flexibility, by describing a predictive test for programming aptitude.

2 Related work

There are two obvious directions of research in the area of first programming teaching and learning. Either we look at the learner to try to find out what they are doing and thinking, or we look at the teacher to see what they are trying to do and with which and to whom. The first kind of study is what is reported in this paper, but the second predominates in the literature.

Learners are hard to experiment on, but teachers make excellent subject: they are keen to change what they teach, how they teach it, when they teach it, how they assess their students' performance, and we don't know what else. On the other hand it is very difficult to research the results of experiments on teachers: for ethical reasons there are rarely if ever control groups, the Hawthorn effect dominates, and all you get is a weak result that when when we did things this way we think our results got better: the sort of result that crumbles the first time it meets the sunlight.

Since many of the researchers are programmers themselves, there is a great temptation to try technological fixes, and the simplest fix is to try a new programming language, sometimes even one devised and implemented on the spot by the researchers themselves. The plains of history are littered with the whitening skulls of studies which tried to show that the One Language That All Can Understand had been discovered: Prolog, Logo, Miranda, Pascal and even C were once capering through the long grass under the fond gaze of educationalists. Java is merely the latest in a long line of delusory mirages.

The other kind of technical fix, in these days of graphical user interfaces, is the IDE (integrated development environment). Surely, if we make programming a bit more point-and-click then it will get easier. Well, no it doesn't, actually. The skulls pile up pretty high there, because IDEs are good MSc and PhD projects (programming languages are a bit harder).

Some of the published research is so far off beam that it's hard to know what the researchers were thinking. Dyck and Mayer, for example, in [16], looked at the comprehension of statements in BASIC and what they believed were similar statements in English. The subjects who were examined on BASIC were familiar with BASIC, and those who dealt with English statements were familiar with English. They found a correlation between the time it took their subjects to comprehend similar statements in the two languages.

Soloway was a pioneer in the area, counting, cataloguing and classifying novices' mistakes and misconceptions. Adelson and Soloway, in [1], reported that domain experience affects novices' ability to understand the sort of programming problems that they are asked to solve. Bonar and Soloway, in [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 the startling theory that having prior knowledge of one programming language has a negative impact on novices' attempts to program in a second language. Soloway and Spohrer, in [38], looked at background knowledge. They opined that skill in natural language had a great deal of impact on novices' conceptions and misconceptions of programming.

Putnam and others, in [35], found that novices' misconceptions about the capabilities of computers could have a massive negative impact on their success at programming. Many students apparently tried to use meaningful names for their variables, hoping that the machine would be able to read and understand those names and so perceive their intentions.

Kessler and Anderson, in [26], looked at students learning recursion and iteration. They came to the distinctly underwhelming conclusion that those who didn't have the first idea what they were about didn't have and couldn't make good learning strategies either.

Du Boulay, in [4], tried to catalogue the difficulties that students experienced. He identified orientation (what programming is), machinery (understanding what a computer is), notation, structure in formal statements, and pragmatics (how to debug). He also categorised students' mistakes into misapplications of analogies, overgeneralisations, and inappropriate interactions of different parts of a program.

Pennington, in [33], looked at the way that expert programmers understand problem domains and programs. 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 it.

Perkins and others, in [34], thought that they had found alternative strategies in learners. Some they called "stoppers", who appeared to give up at the first difficulty. Others, called "movers" seemed to use natural language knowledge to get beyond an impasse.

Van Someren, in [39], looked at novices learning Prolog. 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 difficulties.

Mayer, in [29], noticed that the formal nature of programming (he called it syntactic knowledge) is the difficult bit. He believed that experts were able to think semantically. In [30] he showed that people who know how their programs work do better than those who do not. Canas and others, in [13], came to the same sort of conclusion.

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

Hewett, in [20], tried to use the study of psychology to motivate learning to program. His subjects enjoyed the experience, and no doubt learned some psychology along the way.

Linn and Dalbey, in [27], castigated teachers who go too fast for some of the members of their class.

Murnane, in [32], tried to relate programming to psychological theories, specifically Chomsky's notion of natural language acquisition and Piaget's theories of education. Winslow, in [42] tried to distinguish between novices and experts: experts can solve problems better than novices, it would seem.

Brusilovsky and others, in [12], described the language KyMir, now lost on the great plain of dead languages.

Amongst the vast collection of work trying out programming tools and IDEs are Giannotti [17], Ramadhan [37], Boyle and Margetts [11], Boyle et al [7], Boyle and Green [8], Boyle and others [9], Gray et al [18], Quinn [36] and Chalk et al [14]. Unfortunately none of it amounts to anything at all.

Thomas Green, in [19], put forward the notion of *cognitive dimensions* to characterise programming languages and programming problems. Like Detienne [15], he rejects the notion that programming languages can be 'natural': they are necessarily artificial, formal and meaningless.

Johnson-Laird has contributed the idea of *mental models* to the study of people's competence in deductive reasoning. With Wason [41] he discovered that people make systematic errors in reasoning, and that they tend to draw on the content of premises to inform their conclusion, in ways that aren't deductively valid [40]. 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 theory.

Later, Johnson-Laird put forward the theory that individuals reason by carrying out three fundamental steps [21]:

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 [25]. Johnson-Laird concluded [22] that comprehension is a process of constructing a mental model, and set out his theory in an influential book [23]. Since then he has applied the idea to reasoning about Boolean circuitry [3] and to reasoning in modal logic [24].

And so on and on. Apart from Green and Johnson-Laird, there are no theories. Apart from Soloway’s observation that prior knowledge of a programming language is a bad thing, Ahmazadeh et al’s that good programmers aren’t always good debuggers, and Pennington’s finding that programmers don’t write top-down, there are no results at all. There is absolutely nothing to show from all those IDEs and tools, and all those fancy programming languages have vanished without trace. It’s a pretty dismal picture. No wonder teachers become dispirited.

3 The experiment

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

- assignment and sequence;
- recursion / iteration;
- concurrency.

Few programmers ever reach concurrency, which is the highest hurdle of the three. Recursion is conceptually difficult, and 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.

We devised a test, with questions as illustrated in figure 1. Each question gave a sample Java program, declaring a couple of variables and executing one, two or three assignment instructions. We had some notion of the mental models that a student might use to answer our questions. In particular we believed that they would read the equality sign ‘=’ as a description of equality. We expected that, given earlier instruction in arithmetic and algebra, they would have an idea

<p>1. Read the following statements and tick the correct answer in the front column.</p> <pre>int a = 10; int b = 20; a = b;</pre>	<p>The new values of a and b are:</p> <p><input type="checkbox"/> a = 30 b = 0</p> <p><input type="checkbox"/> a = 30 b = 20</p> <p><input type="checkbox"/> a = 20 b = 0</p> <p><input type="checkbox"/> a = 20 b = 20</p> <p><input type="checkbox"/> a = 10 b = 10</p> <p><input type="checkbox"/> a = 10 b = 20</p> <p><input type="checkbox"/> a = 20 b = 10</p> <p><input type="checkbox"/> a = 0 b = 10</p> <p><input type="checkbox"/> If none, give the correct values: a = b =</p>	
--	---	--

Figure 1: A sample test question

that equality was related to substitution of equals for equals, and we invented our mental models accordingly. (Note that in this we were assisted by the decision of the Java designers, which as computer scientists we would normally excoriate, to use the equality sign to indicate assignment.) The middle column gives a multiple-choice list of the alternative answers which correspond to the models we had decided upon.² The blank column was for rough work: we found the marks that the subjects made here occasionally helpful.

Our intention was to observe the mental models that students used when thinking about assignment instructions. 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 program. We were able to make our programs look quite like the mathematical formulae that our subjects would have encountered in school arithmetic and algebra lessons, and we were therefore able to predict with considerable accuracy the mental models that very many of our subjects employed.

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. We expected 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 works. So we planned to administer the questionnaire just after the students had begun 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 normal end-of-course examination.

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.

² By accident we had missed out two models which, by symmetry, we ought to have included. In figure 1 there should have been ten elements in the list, rather than eight, but in the event our error didn’t obscure our result.

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 ($b := a+b$ – 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).

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 an adult education course at Barnet College *before they had received any programming teaching whatsoever*. Those students had no particular pattern of age, gender and educational background. Though we did no detailed interviews, we believe that none had any previous contact with programming, and that all or almost all had some school mathematics teaching that made 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; again, we believe, the students 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 over 60 subjects.

3.2 Second administration

The test was administered again to the same subjects (with the exception of the foundation year) after about the first three weeks of teaching. Anonymisation introduced difficulties, because some students had forgotten their nicknames. Comparison of handwriting and some inspired guesses mean that we got almost all of them.

3.3 Third administration

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

3.4 Correlation

We correlated the results of the first and second administrations, using the code of anonymity. Because it would have been difficult to get hold of the detailed exam results, we averaged the results of a couple of conventional in-course exams that one of us had already administered as part of the normal teaching process. With our subjects' assistance we broke the code of anonymity, and correlated the text and in-course exam results, as well as the crude pass/fail results of the official examination, which were publicly available.

4 Results

It seems absurd to administer a test to subjects who can have no idea what the questions meant. 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. It appears to us that for the most part they used models which were about change.

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

We did not interview our subjects to determine anything about their group membership. 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 those choices mean. We have no information about how group membership correlates with earlier education, with age, with gender or with 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

Table 2: Stable group membership between first and second administrations

	Consistent (A2)	Inconsistent (A2)	Total
Consistent (A1)	26	1	27
Inconsistent (A1)	11	13	24
Blank (A1)	5	5	10

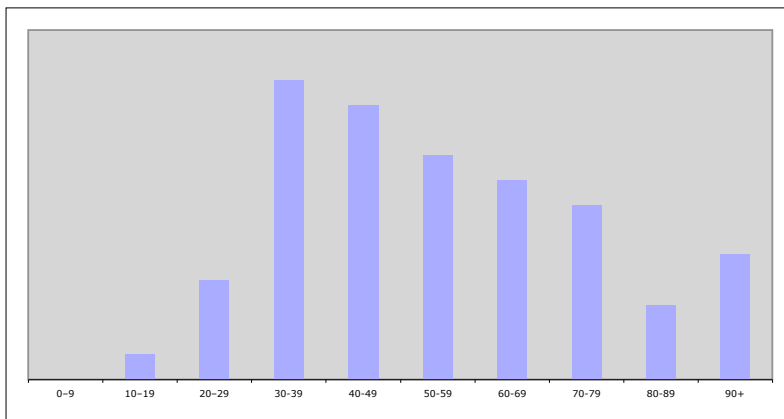


Figure 2: In-course exam results

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. Nobody, so far, has predicted that the consistent group would be the most successful. Remarkably, it is the consistent group that is successful. We speculate on the reasons in section 5.

Table 2 shows our first result. There is essentially no movement from consistent to inconsistent (one person, who did not otherwise take part in the course, did switch but we doubt that he took the second administration seriously). There is some movement – almost half – from inconsistent or blank to consistent. Everyone who had been blank in the first test became either consistent or inconsistent. Despite the apparent mixing, a χ^2 test assures us that, with $p \leq 0.0001$, the two populations are distinct.

The in-course exam results are shown in figure 2: hardly a classic double-hump curve, but by no means a normal curve either. When redrawn in figure 3 to distinguish those who had been

Table 3: End-of-course exam pass/fail statistics

	Consistent	Inconsistent/Blank	Total
Pass	23	20	43
Fail	2	16	18
Total	25	36	61

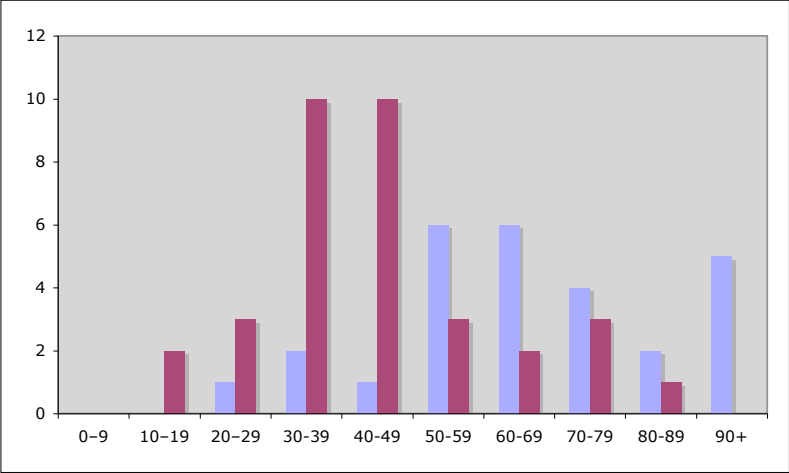


Figure 3: Consistent (blue) and inconsistent (red) in-course exam results

Table 4: In-course exam pass/fail statistics (pass mark 50%)

	Consistent	Inconsistent/Blank	Total
Pass	21	9	30
Fail	4	27	31
Total	25	36	61

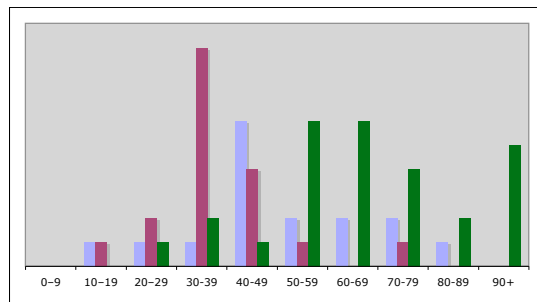


Figure 4: Non-switchers (red), switchers (blue) and consistent (green) in-course exam results

consistent in the first test (blue) from those who had been inconsistent or blank (red) it is clear that we do have a double hump. Drawing up a table to distinguish those who passed from those who failed in the final official examination against performance in our first test gives us table 3. The examination is not very good at distinguishing the populations, but our test is robust: a χ^2 test gives us $p \leq 0.01$. If we look again at figure 3 and select a pass mark of 50% (chosen before we saw the double-hump data, honestly!) we get table 4: a much starker statement of the position.

The switchers in table 2 are not entirely indistinguishable from those who stay put, but they are much more like them than they are like the consistent group. Figure 4 compares the three populations. The switchers (blue) are a little bit farther to the right – that is, get slightly higher marks – than the non-switchers, but they are clearly distinguishable from the consistent group (green).

We believe that our data analysis supports the notion that there are indeed two groups with different levels of performance. Because of the nature of the examinations, which were intentionally asking very simple questions and were designed so that a guesser could score substantially,³ those who can program weren't very much separated from those who could not. We must turn to other data to see more separation: see section 6.

5 Speculation

We believe that we have confirmed every programming teacher's belief that there are two populations in initial programming courses, and found a test which distinguishes them, separating those subjects who are capable of success – the consistent group – from those who are not – the rest. Since nobody ever leaves the consistent group, 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 sort of

³ Another effect of misguided Quality Assurance, and once again that must be by the by.

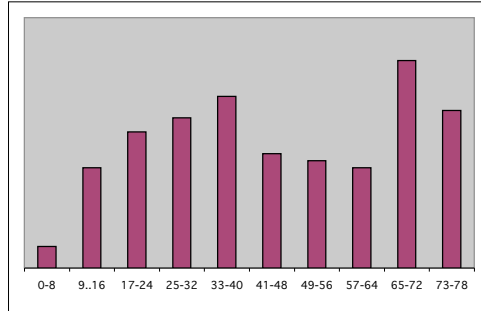


Figure 5: Double-humped exam results in formal proof

predictive success.

It has taken us some time to come to terms with this fact and 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 two groups is their different attitudes to meaningfulness.

Formal logical proofs, and therefore programs – which are 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 the problem seems to mean, the machine will blindly follow its meaningless rules and come to some meaningless conclusion. In behaving consistently 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 meaningfulness, and rejects it.

If we are right, and we stress again that we do not yet have sufficient data to make the assertion, then it is impossible to teach programming to the inconsistent and blank group. They will never come to terms with the meaningfulness of mechanical computation, and so it is cruel to make them try. On the other hand, it is pointless to teach programming, at least so far as the assignment and sequence topic is concerned, to the consistent group. All they need is to be pointed at the correct model of assignment, and they are off.

Programming teachers have always suspected that teaching programming is impossible for much of the class, and we have added to that that it is pointless for the rest. A dark cloud, indeed, but one with a substantial silver lining: there is a substantial minority that is destined to be capable of learning to program.

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 written a proof calculator, Jape [43], and hoped that with mechanical assistance students could at least learn to make formal proofs. He speculated that the size of programming languages might confuse many students: 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 (you need to know the library before you can do anything in Java, so you need to read the library specification

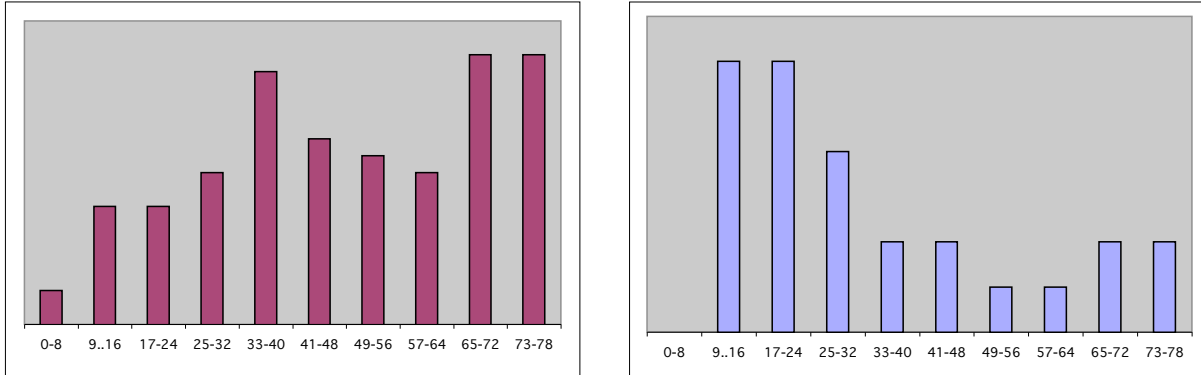


Figure 6: High achievers' (red) and low achievers' (blue) exam results in formal proof

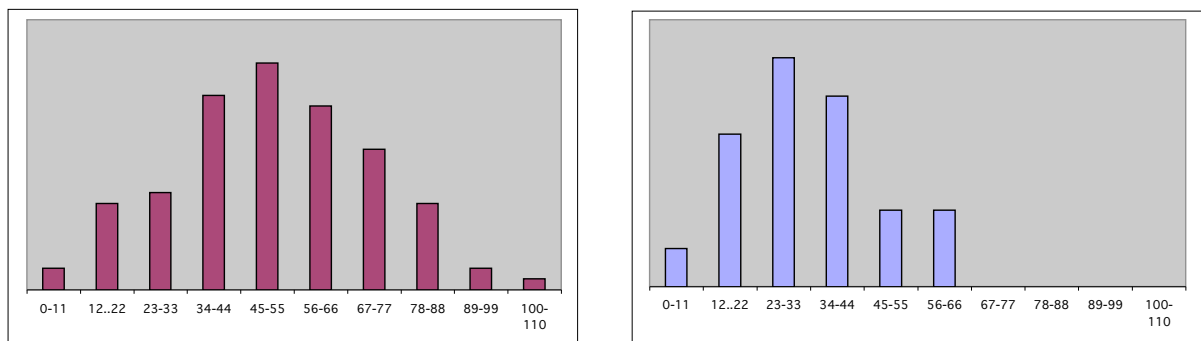


Figure 7: High achievers' (red) and low achievers' (blue) exam results in informal disproof

before you start). 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. Figure 5 shows the results from a test on formal logical proof. Unlike the in-course 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 our test was developed, so we can't analyse it further, but it is rather nice evidence of the double-hump phenomenon.

There is more. Within the course there were several sub-groups, due to the vagaries of university admissions procedures. The group called G500 were admitted to a computer science course and had higher than average A-level scores within the population of students admitted to UK universities. The group called GG51 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. When we separate out the scores of G500 students and GG51 students, in figure 6, we can see that the G500s have a larger success hump and a failure hump centred on 33-40 – about 50% – whereas the GG51s have a smaller success hump and a proportionally much larger failure hump at about 16 – about 20%. (These two graphs don't add up to give figure 5 because there were other groups on the course.)

We speculate from this data that the proportion of programmers in the population varies according

to ‘intelligence’ or educational attainment at least.

There is still more. The *same students*, on the *same course*, when given an nonformal 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 7 shows the data. It also shows the difference in attainment: the G500 hump is well within the passing zone, if a little bit lower than the administrator-desired 55%, whereas the GG51 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 a test for success in a first programming course. We don’t understand how it works any more than you do. An enormous space of new problems has opened up before us all.

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] T Boyle, J Gray, B Wendi, and M. Davies. Talking the plunge with clem: the design and evaluation of a large scale cal system. *Computers and Education*, 22:19–26, 1994.
- [8] T. Boyle and Thomas Green. Build your own virtual computer : a computer simulation using an active learning approach. In P. Brusilovsky, editor, *East-West Conference on Multimedia, Hypermedia and Virtual Reality (MHVR '94)*, Moscow, 1994.
- [9] T Boyle, B Stevens-Wood, F Zhu, and A Tika. Structured learning in virtual environments. *Computers and Education*, 261(3):41–49, 1996.
- [10] 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.

- [11] Tom Boyle and Sue Margetts. The core guided discovery approach to acquiring programming skills. *Comput. Educ.*, 18(1–3):127–133, 1992.
- [12] P Brusilovsky, A Kouchnirenko, P Miller, and Tomek. Teaching programming to novices: A review of approaches and tools. In *World Conference on Educational Multimedia and Hypermedia (ED-MEDIA 94)*, pages 103–110, Vancouver, 1994.
- [13] J. J Canas, M. T Bajo, and P Gonzalvo. Mental models and computer programming. *Journal of Human-Computer Studies*, 40(5):795–811, 1994.
- [14] Peter Chalk, Claire Bradley, and Poppy Pickard. Designing and evaluating learning objects for introductory programming education. In *8th annual conference on Innovation and technology in computer science education*, page 240, Thessaloniki, Greece, 2003. ACM Press.
- [15] 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.
- [16] Jennifer L Dyck and Richard E. Mayer. Basic versus natural language: is there one underlying comprehension process? In *SIGCHI conference on Human factors in computing systems*, pages 221–223, San Francisco, Ca, 1985. ACM Press.
- [17] Elena Giannotti. Algorithm animator: a tool for programming learning. In *18th SIGCSE technical symposium on Computer science education*, pages 303–314, St. Louis, Missouri, 1987. ACM Press.
- [18] John Gray, Tom Boyle, and Colin Smith. A constructivist learning environment implemented in java. In *6th annual conference on the teaching of computing and 3rd annual conference on Integrating technology into computer science education: Changing the delivery of computer science education*, pages 94–97, Dublin City Univ., Ireland, 1998. ACM Press.
- [19] Thomas Green. Cognitive approach to software comprehension: Results, gaps and introduction. In *Workshop on Experimental Psychology in Software Comprehension Studies*, University of Limerick, 1997.
- [20] Thomas T. Hewett. An undergraduate course in software psychology. *SIGCHI Bull.*, 18(3):43–49, 1987.
- [21] P.N. Johnson-Laird. Models of deduction. In R Falmagne, editor, *Reasoning: Representation and Process*. Erlbaum, Springdale, NJ, 1975.
- [22] P.N. Johnson-Laird. Comprehension as the construction of mental models. *Philosophical Transactions of the Royal Society, Series B*, 295:353–374, 1981.
- [23] P.N. Johnson-Laird. *Mental Models*. Cambridge University Press, Cambridge, 1983.
- [24] 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.
- [25] P.N Johnson-Laird and M.J. Steedman. The psychology of syllogisms. *Cognitive Psychology*, 10:64–99, 1978.
- [26] C. M Kessler and J. R Anderson. Learning flow of control: Recursive and iterative procedures. *Human-Computer Interaction*, 2:135–166, 1986.

- [27] M Linn and J Dalbey. Cognitive consequences of programming instruction. In E. Soloway and Spohrer, editors, *Studying the Novice Programmer*, pages 57–82. Publisher: Lawrence Erlbaum Associates, 1989.
- [28] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Mostr, Kate Sanders, Otto Sepp al a, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers, 2004.
- [29] Richard E Mayer. The psychology of how novices learn computer programming. In E. Soloway Spohre and J.C., editors, *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, 1989.
- [30] Richard E Mayer. *Thinking, Problem Solving, Cognition*. W. H. Freeman and Company Second Edition ISBN 0716722151, New York, 2 edition, 1992.
- [31] 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.
- [32] J Murnane. The psychology of computer languages for introductory programming courses. *New Ideas in Psychology*, 11(2):213–228, 1993.
- [33] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19:295–341, 1987.
- [34] 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.
- [35] 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.
- [36] Alexander Quinn. An interrogative approach to novice programming, 2002.
- [37] Haider Ramadhan. An intelligent discovery programming system. In *ACM/SIGAPP Symposium on Applied computing: technological challenges of the 1990s*, pages 149–159, Kansas City, Missouri, United States, 1992. ACM Press.
- [38] 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.
- [39] Maarten W van Someren. What’s wrong? understanding beginners’ problems with prolog. *Instructional Science*, 19(4/5):256–282, 1990.
- [40] P. C. Wason and P.N. Johnson-Laird. *Psychology of Reasoning: Structure and Content*. Batsford, London, 1972.

- [41] P.C. Wason and P.N. Johnson-Laird. *Thinking and Reasoning*. Harmondsworth: Penguin, 1968.
- [42] Leon E Winslow. Programming pedagogy: a psychological overview. *SIGCSE Bull.*, 28(3):17–22, 1996.
- [43] Richard Bornat (with Bernard Sufrin). Jape software. <http://www.jape.org.ac.uk>.