Naturally Occurring Data as Research Instrument: Analyzing Examination Responses to Study the Novice Programmer

Tony Clear
School of Comp. & Math. Sciences
AUT University
New Zealand
tony.clear@aut.ac.nz

Raymond Lister
Department of Computer Science
University of British Columbia
Canada
raymond@it.uts.edu.au

Simon
Faculty of Science & IT
University of Newcastle
Australia
simon@newcastle.edu.au

Dennis J Bouvier
Department of Computer Science
Southern Illinois Univ. Edwardsville
United States of America
dbouvie@siue.edu

Paul Carter
Department of Computer Science
University of British Columbia
Canada
pcarter@cs.ubc.ca

Anna Eckerdal
Dept of Information Technology
Uppsala University
Sweden
Anna.Eckerdal@it.uu.se

Jana Jacková
Faculty of Manag. Sci. and Informatics
University of Zilina
Slovakia
Jana.Jackova@fri.uniza.sk

Mike Lopez
Manukau Institute of Technology
New Zealand
mike.lopez@manukau.ac.nz

Robert McCartney
Dept Comp. Sci. and Engineering
University of Connecticut
United States of America
robert@cse.uconn.edu

Phil Robbins
School of Comp. & Math. Sciences
AUT University
New Zealand
phil.robbins@aut.ac.nz

Otto Seppälä
Dept. of Computer Science and Engineering
Helsinki University of Technology
Finland
oseppala@cs.hut.fi

Errol Thompson
England
kiwiet@acm.org

ABSTRACT
In New Zealand and Australia, the BRACElet project has been investigating students' acquisition of programming skills in introductory programming courses. The project has explored students' skills in basic syntax, tracing code, understanding code, and writing code, seeking to establish the relationships between these skills. This ITiCSE working group report presents the most recent step in the BRACElet project, which includes replication of earlier analysis using a far broader pool of naturally occurring data, refinement of the SOLO taxonomy in code-explaining questions, extension of the taxonomy to code-writing questions, extension of some earlier studies on students' 'doodling' while answering exam questions, and exploration of a further theoretical basis for work that until now has been primarily empirical.

Categories and Subject Descriptors
K.3 [Computers & Education]: Computer & Information Science Education – Computer Science Education.

General Terms
Measurement, Experimentation, Human Factors.

Keywords
Novice programmers, CS1, tracing, comprehension, SOLO taxonomy.

1. INTRODUCTION
The BRACElet project originated in New Zealand in 2004 [15] as a multi-institutional multi-national (MIMN) study into the ways in which programmers, particularly novice programmers, understand how to read and write code. The value of MIMN studies, as noted by Fincher et al. [10], lies in their ability to pool data across institutions in experimental or quasi-experimental studies. Thus broader patterns can be discerned and findings can be derived with greater generalizability than those derived from the all too typical 'lone ranger' studies in computer science education research. The BRACElet studies have combined the work of many collaborators in different institutions and indeed countries, including New Zealand, Australia and the United States of America. The project has evolved over several action research cycles and, as noted by Clear et al. [5], has extended beyond its New Zealand origins to Australia through the award of an Associate Fellowship to Raymond Lister and Jenny Edwards from the Australian Learning and Teaching Council (formerly the

Carrick Institute). This working group represents a further extension of the BRACElet project to a wider international team of collaborators, but remains very much in keeping with the BRACElet spirit of collegial, multi-institutional research in a way that builds on prior Computer Science Education research development initiatives such as 'Bootstrapping' and 'BRACE' [10]. The project includes a mix of novice, intermediate and more senior researchers, most of whom are actively engaged in teaching programming courses, and thus remains close to practice and practitioner concerns while building the skills of the researchers involved. BRACElet has resulted in more than thirty publications to date, (co)written by more than 20 different authors, and is grounded in data arising largely from standard teaching and assessment practices.

1.1 The Value of Examination Data

Early BRACElet studies [15] collected data from students in examination conditions, but not necessarily in examinations. In some cases, the data collection was independent of any formal assessment in the courses the students were studying or had studied.

More recently the project has moved to a stronger dependence on analysis of students' examination answers, a move that offers considerable benefits.

First, participation in the project costs the students nothing. They do not have to set aside extra time to participate, they do not have to spend extra time preparing, they do not have to travel for a one-off appointment. They are already required to do all of these things for the examination. The data is thus naturally occurring: it already exists by virtue of the examination process, and simply requires collection and analysis.

Second, data collection has a minimal cost to the researcher. There are no special appointments, no additional materials, no cost of organization. There is the cost of ensuring that the examination includes questions that are suited to the particular analysis being proposed, but this is clearly less than the cost of designing and conducting an explicit study.

Third, it can sometimes be easier to acquire ethics approval for research involving naturally occurring data than for research involving explicit data collection. Some institutions in some countries are able to directly approve such projects. Even where the individual consent of each participant is still required, students appear to be more willing to consent to the use of their data when it comes at no cost to them.

Fourth, collection of exam data brings a reasonable assurance that all participants are at about the same phase of their learning. They have all just completed the same course, they have all had the opportunity to revise their knowledge in preparation for the exam.

Fifth, the goals of this research are strongly congruent with those of end-of-course examinations. The examination is meant to assess the extent to which students have acquired the knowledge imparted in the course. It would seem wasteful not to then use their answers to address research questions revolving around what and how much they have learned and how they have learned it. The use of this data ensures that the project remains close to pedagogical practice.

When including research-specific questions in examinations, it is vital to ensure that they are still valid examination questions, that their answers will contribute to a summative assessment of the student's learning. Fortunately, because of the close match between the goals of the assessment and the goals of the research, this is not generally a problem.

1.2 A Possible Hierarchy of Programming Skills

Early BRACElet papers [17, 40] reported on a study in which students in an end-of-semester exam were given a question beginning "In plain English, explain what the following segment of Java [or Pascal or C++] code does". A correlation was found between how well students answered that type of question and how well they performed on other programming-related tasks. A conclusion of those early papers was that there is a hierarchy of programming skills, and that the ability to provide such a summary of a piece of code – to 'see the forest and not just the trees' – is an intermediate skill in that hierarchy. Much BRACElet work since then has further explored this postulated hierarchy.

Philpott et al. [25] reported results indicating that the ability to manually execute ('trace' or 'desk check') code is lower in the hierarchy than the ability to explain code. Sheard et al. [30] found that the ability of students to explain code correlates positively with their ability to write code.

Analyzing student responses to an end-of-first-semester exam, Lopez et al. [18] used stepwise regression to construct a hierarchical path diagram in which basic knowledge occupied the lowest level and writing code occupied the highest. In the intermediate levels of the regression were the ability to trace non-iterative code, then the ability to trace iterative code and the ability to provide a summary for 'explain in plain English' questions. A recent follow-up study at an Australian university [16] produced results consistent with this finding.

Belief in the importance of tracing skills and of skills similar to explaining can be found in the earlier literature on novice programmers. Perkins and Martin [24] discussed the importance and role of tracing as a debugging skill. Soloway [32] suggested that skilled programmers carry out frequent 'mental simulations' of their code, which can be more abstract than tracing the code, and he advocated the explicit teaching of mental simulations to students.

BRACElet continues to explore the possibility of this hierarchy, in the belief that, if firmly established, it could be of benefit both as a diagnostic tool and as a pedagogical guideline.

1.3 The Common Core

The BRACElet 2009.1 (Wellington) specification [41] defines a common core of question types, with the goal of enabling cross-site consistency in the data captured and in its subsequent analysis. The core consists of exam-type questions in three categories: Basic Knowledge and Skills, Reading / Understanding, and Writing. Participants in this working group were required to collect data based on the common core: not to use exactly the same questions, but to use questions that fit into each of these three categories.

1.3.1 Basic Knowledge and Skills

Basic knowledge and skills questions require students to trace or hand-execute code. The questions establish that students understand the programming constructs (for example, how an 'if'

statement or a ‘while’ loop works), and that students can reliably track variable updates while tracing through code.

1.3.2 Reading / Understanding

Reading and understanding questions include ‘explain in plain English’ questions and questions known as ‘Parsons puzzles’*, where students are given lines of code in random order and are required to put them into the correct order [9, 21]. The purpose of this part of the common core is to establish whether students can see how the parts of a small program work together to perform the overall computation – to see the forest, not just the trees. We do not suggest that these two question types are equivalent, but at present we place them together following a postulate [42] that the distinct skills they require are both intermediate between tracing and writing skills.

1.3.3 Writing

Writing questions require students to write code. When considering the postulated hierarchy of skills, students’ performance on code-writing questions is the dependent variable. In more general terms, the ability to write program code is what we aim to teach, so anything else that we can discover about students’ acquisition of skills must ultimately be considered in the light of their ability to write code.

1.4 BRACElet and the SOLO Taxonomy

The BRACElet project has for some time [40] been classifying code-reading questions with the SOLO taxonomy [2, 3]. Thompson [37] explains that “SOLO stands for Structure of the Observed learning Outcome. It is based on a quantitative measure (a change in the amount of detail learnt) and a qualitative measure (the integration of the detail into a structural pattern). The lower levels focus on quantity (the amount the learner knows) while the higher levels focus on the integration, the development of relationships between the details and other concepts outside the learning domain.” Using this taxonomy, students’ answers are classified not so much according to their correctness as according to the level of integration that they demonstrate, the idea being that so long as it is actually correct, a more integrated answer is a more convincing demonstration that the student has understood the code.

The application of the original SOLO taxonomy to plain-language explanations of program code is not entirely intuitive, and the BRACElet project has added a number of intermediate levels. Part of the problem is that the notion of correctness in explaining program code is somewhat more strict than the same notion in explaining, say, the workings of democracy. Therefore members have felt that even when an explanation is substantially correct, it is important that a classification recognize whatever errors it might encompass. A workshop following the ICER 2008 conference in Australia proposed the version of the levels presented in Table 1. This same workshop observed that while agreements on ratings appeared to be quite consistent after a process of joint categorization in small groups, the levels should still be regarded as in a state of flux; it therefore recommended that future BRACElet workshops discuss the levels in the context of their own data and determine whether further refinements might be warranted.

Table 1: The SOLO levels as applied to explaining code, as at mid-2008

SOLO category	Description
Relational (R)	A summary of what the code does in terms of its purpose (the ‘forest’)
Relational Error (RE)	A summary of what the code does in terms of its purpose, but with some minor error
Multistructural (M)	A line-by-line description of all the code (the ‘trees’)
Multistructural Error (ME)	A line-by-line description of most of the code, with some minor errors
Unistructural (U)	A description of one part of the code
Prestructural (P)	Substantially lacks knowledge of programming constructs or is unrelated to the question

1.5 Overview of Working Group

As ITiCSE attracts participants from many countries, it was expected that the students whose data was brought to the working group would encompass a broad range of programming abilities, cultural backgrounds, and approaches to teaching programming. Additional contextual variations would include the programming language of instruction, the natural language of instruction, and many other academic variables such as class size, laboratory setting, etc. Thus the working group would provide a test to the generality of the prior findings of the BRACElet project, which are based on data collected from students at a small number of New Zealand and Australian universities. At the outset of the working group meetings, the goals could be encapsulated in the following questions:

- How does the work to date of the BRACElet project tie in with existing theoretical research on students’ acquisition of skills?
- Does analysis of the data brought to the working group support prior BRACElet findings, which were generally based on smaller sets of data?
- Is it informative to classify students’ answers to code-explaining questions according to the SOLO taxonomy, and does this classification give rise to useful results?
- Can the SOLO taxonomy be usefully extended to cover not just code-explaining questions but code-writing questions?
- What else might emerge from consideration of this wealth of data in the intense setting of the working group and the diverse research focuses of the participants?

2. WORKING GROUP DATA

The data brought by working group members consisted of exam questions and students’ answers in programming courses offered

---

* When introduced, these were called *Parson’s puzzles* [21]. The apostrophe is confusing, as the questions were named not after a parson or a person called Parson, but after a person called Parsons. We follow subsequent literature [9] in calling them Parsons puzzles or Parsons problems, meaning puzzles or problems in the style of Parsons.

Table 2: Summary of the seven datasets analyzed in various parts of this report

Dataset	PA	PD	PF0	PF1	PK	PM	PN
Students	330	97	43	49	93	76	582
Level of course	1	1	0.5/1	2	1	1 (sem 2)	1
Language	C	Visual Basic	C#	Perl	Java	Pascal	Python
Tracing questions	yes	yes	yes	yes	yes	yes	yes
Explaining questions	no	yes	yes	yes	yes	yes	yes
Parsons problems	yes	no	yes	yes	yes	yes	no
Writing questions	yes	yes	no	no	yes	yes	yes

in Australia, Canada, Finland, New Zealand, Singapore, Slovakia, and the United States. Where applicable, members had obtained ethics approval to use their students' work for research purposes.

Nine exams were provided, eight from introductory programming courses and one from an advanced data structures course. The answers of nearly 1300 students were available for analysis. The programming languages covered were C, C++, C#, Java, Pascal, Perl, Python, and Visual Basic. All of the exams included code-tracing questions, most included code-explaining and code-writing questions, and most included Parsons questions. Table 2 briefly summarizes the seven datasets that have been used for analysis in this report.

While some of the working group members carried out empirical analysis of one or more datasets, others conducted theoretical work. The related theory base is rich and has its roots in general education, mathematics education, computer science education, and psychological theories, which are summarized in the next section.

3. PROCESSES AND OBJECTS

Although it began from a fairly empirical basis, the BRACElet project ties in well with existing theoretical research. There is a large body of research in mathematics education in which knowledge is divided into two main types, often referred to as *conceptual* and *procedural* knowledge. McCormick [20] writes that the terms conceptual and procedural knowledge relate to "a familiar debate in education, namely that of the contrast of content and process (p149) … In mathematics education the argument has been about 'skills versus understanding'." Early work in this tradition includes that of Hiebert and Lefevre [14] who defined *conceptual knowledge* as "rich in relationships…a connected web of knowledge, a network in which the linking relationships are as prominent as the discrete pieces of information" (pp3-4), and *procedural knowledge* as "made up of two distinct parts … the formal language, or symbol representation system, of mathematics … [and] the algorithms, or rules, for completing mathematical tasks" (p6).

Different researchers have used somewhat different terminology when exploring these shifts in perspective. In order to provide a common frame of reference we first consider the overall model as presented by Sfard [29], who describes two alternative views of knowledge:

- Operational or process understanding considers how something can be computed; the concept is regarded as an algorithm for manipulating things.
- Structural or object understanding describes a concept by its properties, treating it as a single entity.

A process view allows a student to apply the concept to data, while an object view allows a student to reason about the concept, to treat it as data.

Consider the example of *function*. The operational view of function is the computational process of mapping from x to y: for a function $y = 3x^2$, it would be the process of squaring x and multiplying the result by 3 to obtain y. A structural view of the function might be a set of ordered pairs or a plot of x against y.

Sfard provides a three-phase mechanism for concept formation, from process to object understanding. These phases are:

- *Interiorization*: the student becomes familiar with applying the process to data.
- *Condensation*: the student abstracts the process into more manageable chunks. Ultimately the student may abstract the process into its input-output behavior (in computing terms), but will still view the concept as an algorithmic process.
- *Reification*: the student views the concept as a unified entity. The concept is understood by its characteristics, and can be manipulated as a primitive object in other concepts. This is the most difficult of these transitions, as it involves a transformation of perspective.

These phases build upon one another, so a student who has attained an object understanding still retains the process understanding.

Table 3 shows a mapping between Sfard's terminology, those used by other authors (Dubinsky, Gray and Tall – after Pegg and Tall [23]), and the corresponding SOLO levels. Dubinsky extends Sfard's model with an additional level, schema, which further abstracts objects. Gray et al. [11] add a new term, procept, being "the amalgam of a *process*, a *concept* output by the process, and a *symbol* that can evoke either process or concept" (p113). Tall [34] develops this further and discusses mathematics students' progress through the five stages of pre-procedure, procedure, multi-procedure, process, and procept.

Baroody et al. [1] give a brief overview of work in this field. With reference to Star [33], they observe that "each type of knowledge – procedural and conceptual – can have either a superficial or deep quality" (p115).

Star [33] proposes defining conceptual knowledge as "knowledge of concepts or principles" – knowledge that involves relations or connections, but not necessarily rich ones. He defines procedural

Table 3: Comparison of names of stages in Process - Object transition, various authors

Sfard	Dubinsky	Gray and Tall	SOLO
Process	Action	Procedure	Unistructural
(interiorization)			
(condensation)	Process	Process	Multistructural
(reification)			Relational
Object	Object	Procept	
	Schema		Extended abstract

knowledge as "knowledge of procedures" and deep procedural knowledge as involving "comprehension, flexibility, and critical judgment and distinct from (but possibly related to) knowledge of concepts" (p116).

The transition from procedural to structural understanding takes place on a concept-by-concept basis. Importantly, once a concept is reified, it can be used as a primitive in higher-level concept acquisition. Figure 1 illustrates the connection between successive concepts, where the object view of Concept A is used in the process understanding of B, and likewise concept B in C. According to this cycle, the extended abstract SOLO level in Table 3 could also be known as unistructural*: the object understanding becomes a unistructural understanding of the concepts that use it as a primitive component (the asterisk is meant to indicate that it is unistructural in a different concept). Within its own concept, this level of understanding is at least relational; being able to use it in a new concept may raise it to an extended abstract understanding.

The presence of successive concept formation cycles complicates the notion of ordered steps; within a concept, the order is straightforward, as the models assume sequential stages. However, the order of stages from different concepts will reflect the order of the concepts, not the stages. It has also been observed [29] that these cycles are related both internally and sequentially: the desire to use a concept as a concrete object may drive the development of the earlier stages, and the need for a particular primitive in a concept may drive the reification in a previous concept.

In comparing the BRACElet work to this research from mathematics education there is a need for some clarifications. First, we compare skills in computer programming to what is commonly called procedures in mathematics education research. We argue that the two are comparable since they represent the following of a set of step-by-step instructions using the operations of the respective subject areas. Further, the BRACElet project explores skill acquisition while mathematics education research involves concept acquisition. Sfard [28], quoted in Cottrill et al. [8], says:

> "We find in the literature that there is general agreement that process or operational conceptions must precede the development of structural or object notions" (p173).

We believe this to be a fundamental difference between the two subject areas. Computer science has both practical and conceptual

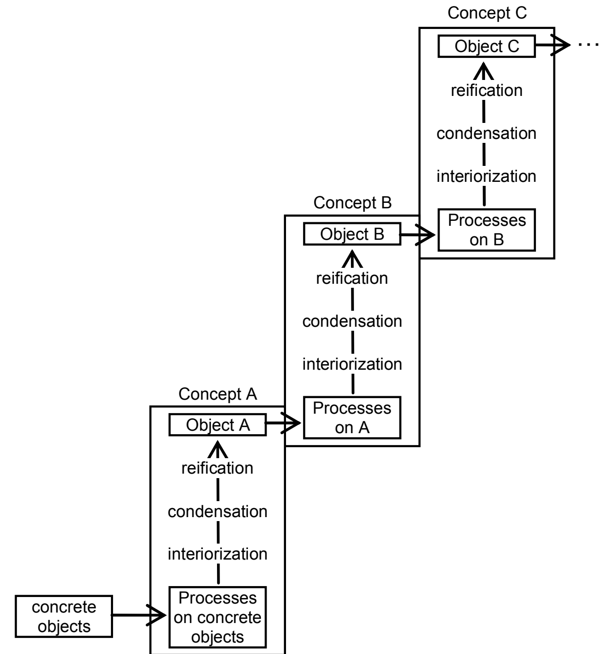

Figure 1: General model of concept formation (after Sfard [29])

learning goals. The skills are not merely ways to reach the more sophisticated conceptual learning goals, but are goals in and of themselves. At the same time, reading, writing, tracing, and explaining code are tools to reach all learning goals, conceptual as well as practical. This difference notwithstanding, we think it is possible and fruitful to relate the mathematics education research findings to the BRACElet research.

In so doing we will start from two broad groups of research, that of Sfard and Dubinsky [29] and that of Gray, Tall, and others [11, 12, 22, 23, 34].

Gray, Tall, and co-authors relate their discussion entirely to students learning mathematical concepts. We argue that the BRACElet work on skills can be related to the work of Gray and colleagues since much of the BRACElet work concerns students' understanding of loops containing if-statements, so these loops represent the concept to be understood. Sfard's work can be related to the BRACElet work since she discusses "the transition from computational operations to abstract objects". However we believe that a major distinction needs to be made between the nature of abstraction in mathematics education and that in computer science education. As noted by Colburn and Shute [7], abstraction in mathematics constitutes 'information neglect' in which key concepts are discarded to enable concentration on the concept at hand. In computer science, by contrast, abstraction is typified by 'information hiding', in which core concepts are encapsulated to provide a base for the next level of thinking. This notion is consistent with Figure 1, in which layers of thought build upon one another. We believe this is close to the BRACElet project's interpretation of the SOLO taxonomy (Section 1.4 and Table 3), that gaining a relational understanding corresponds to Sfard's reification phase, abstracting from a set of instructions to some more encompassing notion of its input/output behavior.

As an example from the present data, the students at one institution (dataset PD) were asked to explain the following code in plain English:

```
strOne = strTitle(0)
For i = 1 To strTitle.Length - 1
    If strTitle(i).Length > strOne.Length Then
                    strOne = strTitle(i)
    End If
Next
```

An answer classified as Relational according to the SOLO taxonomy was that of student PD004:

> "The overall purpose of the code is to find the longest title in the array."

This answer shows not only an understanding of what the code does, but a capacity to discuss the code as a whole, its overall purpose. The student thus demonstrates procedural as well as conceptual understanding, in the terms of the mathematics education language. Using Gray and Tall's terminology, the student has reached the procept level; using Sfard's terminology, the student has reached the reification phase, having abstracted away the details of the process.

An answer to the same question that was classified as Multistructural is that of student PD030:

> "Give strOne the value of strTitle(i) if the length of strTitle is greater than the length of strOne and keep doing it until the length of strOne became the greatest and until the end of the loop."

This student describes what the code does at the instruction level, but without an overall description. This is discussed by Gray and Tall as a procedure, by Sfard as a process, and by Dubinsky as an action.

In conclusion, we see very strong links between accepted theories of learning in mathematics and the SOLO classification that we are applying to learning in computer programming. Yet while we acknowledge the notion of conceptual hierarchies as proposed in the mathematics education literature, the true extent to which they apply to computing education remains open to question. Given the more discrete concept separation of mathematics argued by Colburn and Shute [7], the neatly hierarchical progression for mathematics concepts in Figure 1 may not apply so simply to computing. The encapsulation of modular concepts at the next level via information hiding might suggest that the interiorisation step is not distinct, but is blended in some way with the condensation step at each higher level. Thus while a layered sequence of steps is probably valid, its operation may differ in the computing context. Nonetheless we believe that through these links, mathematics education research can help to inform our own work, and help in developing stronger theoretical understandings of the progressive acquisition of programming knowledge.

4. REPLICATION OF QUT ANALYSIS

Lister et al. [16] analyzed students' answers from an examination at the Queensland Institute of Technology (QUT). After dividing the class into those who did well and those who did not so well on code-tracing questions, code-explaining questions, and the single code-writing question, they conducted pairwise comparisons of the three, concluding that students who can explain code can generally trace code, and students who can write code can generally both trace and explain code. This analysis supports the notion of a hierarchy of programming skills, with tracing as the most elementary skill, explanation as an intermediate skill, and writing as the top of the hierarchy.

The examination scripts brought to the working group were so varied that aggregation of all the student scripts was impractical, so separate replications of the QUT analysis were carried out on datasets PF0, PA, and PM.

4.1 QUT Analysis of Dataset PF0

The students in dataset PF0 were pre-degree students studying a first-semester introductory procedural programming course.

The course provides only a basic introduction to programming covering variables and data types, branching, and one form of iteration (for loops), but not arrays or procedures. Students wrote console applications in Visual C#, all code being written in the Main function.

The code-tracing questions on the exam asked the students to indicate what would be output by 10 pieces of code, ranging from single lines of output code to questions involving for loops. These are the 'tracing' questions in our analysis. We categorized answers to each of the 10 subsections as being fully correct (that is, all parts of the subsection correct) or not.

The code-explaining questions asked the students to explain the purpose of five pieces of code, four of which included a loop. One piece of code was taken from lecture slides and the other four were previously unseen. Questions were worded so that the students had to give a concise explanation rather than try to describe line by line what the code did. As with the tracing questions, we categorized answers as fully correct or not.

Figure 2 shows the number of tracing questions for which students received full marks. This criterion is harsher than the marks themselves, as an answer that earned partial marks (even as many as 3 out of 4 marks) was judged as incorrect. The figure shows that all students answered at least one tracing question completely correctly and just one student provided completely correct responses to all 10 tracing questions.

Informally, the students appear divided into in two groups, those responding correctly to 7 or more tracing questions and those responding correctly to 6 or fewer tracing questions. For the purposes of comparing the students' tracing capability with other capabilities, we designate these the High Tracing Capability (HTC) and Low Tracing Capability (LTC) groups.

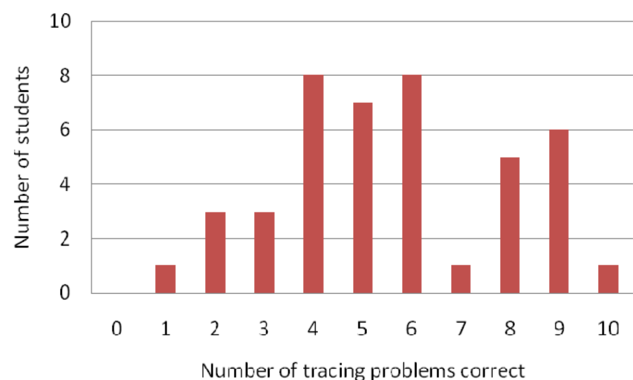

Figure 2: Student performance on tracing questions

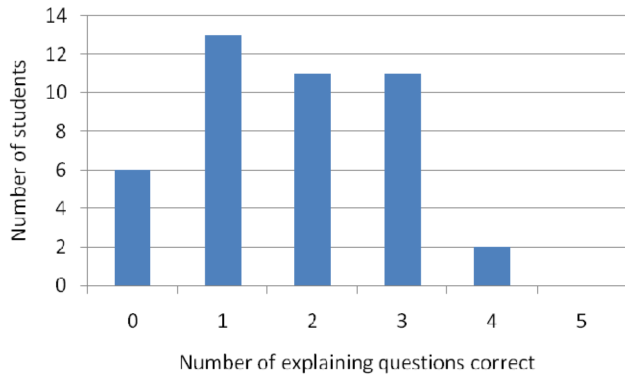

Figure 3: Student performance on 'explain' questions

Figure 3 presents the counts of completely correct responses to code-explaining questions. Again, as with the analysis of the tracing capability, the criterion for counting is full marks for the answer to a question. There are no obvious sub-populations in the code-explaining data.

In Figure 4, the data from Figure 3 is split into the two separate tracing capability groups. In keeping with earlier findings, it is clear that students in the High Tracing Capability group tend to score better on explaining while those in the Low Tracing Capability group tend to perform worse on explaining.

4.2 QUT Analysis of Dataset PA

Dataset PA is from an introductory programming course for engineers. The course is unusual in that the first eight weeks of the course are taught by the Computer Science department while the remaining five weeks are taught by the Computer Engineering department. In addition to the standard elements found in a computer programming course, students are provided with an introduction to a library of functions for interaction with a specialized hardware module.

The exam for this course includes eight code-tracing questions, a single Parsons problem, and three code-writing questions. As answers to code-tracing questions tend to be either right or wrong, our measure of student success on these questions was simply the

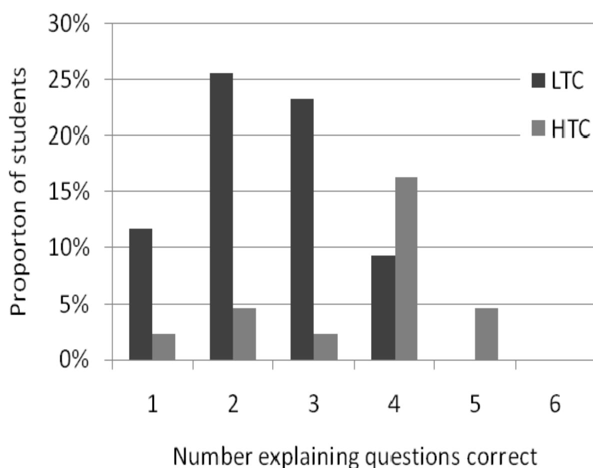

Figure 4: Explaining capability for groups with low and high tracing capability

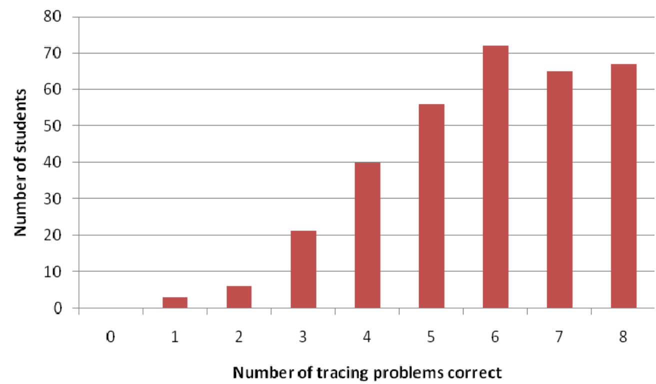

Figure 5: Number of tracing questions correct

number of completely correct answers. On the other hand, code-writing questions admit of an almost continuous range of marks, so our measure of success for these questions is the sum of the students' marks for all three. Figure 5 shows the distribution of correct answers to the code-tracing questions.

Following Lister et al. [16] we now examine pairwise relationships between tracing, Parsons and writing. The correlation coefficient for number of tracing questions correct against the score on the Parsons question is 0.542 (N=330). The correlation coefficient for Parsons against the combined score on the code-writing questions is 0.561 (N=330). Finally, the correlation coefficient for number of tracing questions correct against code writing score is 0.702 (N=330). In all cases, the correlation is significant at the $p < 0.01$ level.

As there was only one Parsons problem we are limited in our analysis of the relationship between this and the other two types of question. We therefore focus our attention on the tracing and writing questions. We designated the High Tracing Capability (HTC) group as those students who scored 7 or 8 (n=132), and the Low Tracing Capability (LTC) group as those who scored 6 or below (n=198). This designation is not entirely arbitrary; it represents what we believe is a reasonable correspondence with the more obvious choice based on Figure 2 in Section 4.1.

For the code-writing questions we divided students into two equal groups based on the median score of 63.9%. Students with the median mark or higher were designated High Writing Capability (HWC), while those below the median were designated Low Writing Capability (LWC).

Table 4 shows the pairwise comparison of the writing and tracing groups. Students who score highly on the tracing questions are more likely to score highly on the writing questions, while those with lower scores on the tracing questions are more likely to achieve lower scores on writing questions. A chi-squared test shows a good correlation, with effect size of 34.4% and $p<0.0001$.

Finally, Figure 6 shows the number of tracing problems correct against average score on the code-writing questions. These results are in keeping with the findings of Lister et al. [16] who found

Table 4: Distribution of students (T and W), N=330

	HWC (N=164)	**LWC** (N=166)
HTC (N=132)	113 (85.6%)	19 (14.4%)
LTC (N=198)	51 (25.8%)	147 (74.2%)

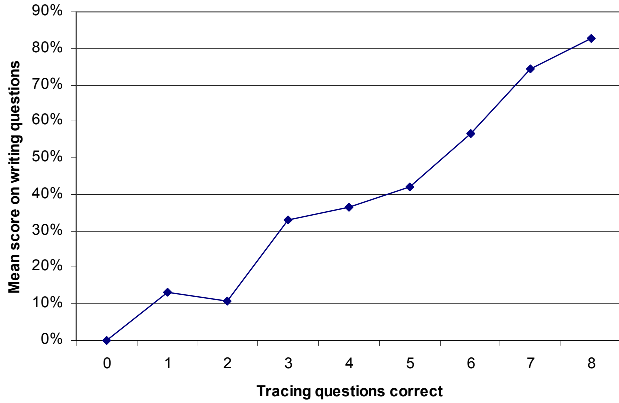

Figure 6: Tracing capability against writing capability

that "most students who scored less than 50% on tracing did poorly on writing". In fact, we see that the average score on writing questions increases almost monotonically with the number of tracing questions correct.

4.3 QUT Analysis of Dataset PM

Dataset PM is from the second half of a two-semester introductory programming course. The course, in an Informatics degree, teaches both procedural and object-oriented programming using Turbo Pascal. The 76 students are not all novices: some have studied programming at secondary school, while others are repeating the course having previously failed it.

The examination includes seven code-tracing questions, five code-explaining questions, four Parsons puzzles, and three code-writing questions.

Four of the code-tracing questions are short-answer questions, so the marking was not simply binary. This being the case, the correctness criterion was defined as a score of 75% or more in a question. Based on that criterion, High and Low Capability groups were designated for Tracing, Explaining, Parsons, and Writing as shown in Table 5.

There are too many pairwise comparisons among the four question types to warrant displaying them all, especially as some of the comparison tables have only single-digit numbers in some cells, but apparent highlights of the comparisons are that:

- Students in the High Tracing Capability group are more likely to be in the High Writing Capability group.
- Students in the High Tracing Capability group are more likely to be in the High Explaining Capability group.
- Students in the High Explaining Capability group are more likely to be in the High Writing Capability group.

Table 5: Designation of capability groups for dataset PM

Type	Questions	Questions correct for	
		Low Capability	High Capability
Tracing	7	0-5	6, 7
Explaining	5	0-2	3-5
Parsons	4	0-2	3, 4
Writing	3	0, 1	2, 3

- Students who are in both the High Tracing Capability group and the High Explaining Capability group are more likely to be in the High Writing Capability group.

4.4 QUT Replications in Summary

In part because of the intense nature of the working group, each of the QUT replications differed slightly in form from the others. Notwithstanding this, each of them tends to support the findings of the original QUT study [16], that students who can explain code can generally trace code, and students who can write code can generally both trace and explain code. This replication thus supports the notion of a hierarchy of skills in learning to program.

5. REPLICATION OF ICER 2008 ANALYSIS

Another study contributing to the notion of a hierarchy of skills was that of Lopez et al. [18]. This study used an empirical stepwise regression approach to construct a possible path diagram of the relationships between skills in basic syntax, tracing, explain in plain English, Parsons puzzles, and code writing. Venables et al. [38] subsequently applied a different method to a new dataset, focusing on the relationships identified by Lopez et al. between the tracing, explaining and writing constructs, and produced results consistent with Lopez et al.

One of the results from the analysis in Lopez et al. was a path diagram derived using stepwise regression, showing the relationships between a set of skills. The most important part of this diagram, the part that includes the relationships between tracing, explaining and writing, is shown in Figure 7. In this path diagram, variables are shown in a titled box containing the variance explained, the adjusted R^2 in square brackets, and the significance of the overall regression. The boxes on the paths show the beta weights (regression coefficients) of the paths, with the semi-partial correlation squared shown underneath. This last represents the unique contribution made to the explanation of the criterion variable over and above that shared with other predictor variables. Thus Tracing and Explaining combined account for 46% of the variability in Writing, while Tracing alone accounts for 39% (the full 46% minus the 7% unique to Explaining), and Explaining alone accounts for 31% (46% minus the 15% unique to Tracing). The direction of the arrows should not be interpreted as evidence of causation.

In this study we used the same core methodology of Lopez et al.

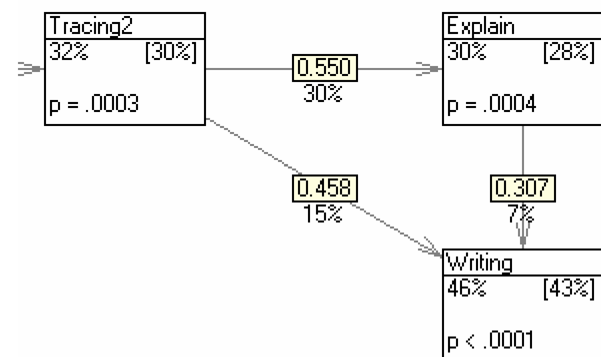

Figure 7: Path diagram (from Lopez et al. [18]) (n=78)

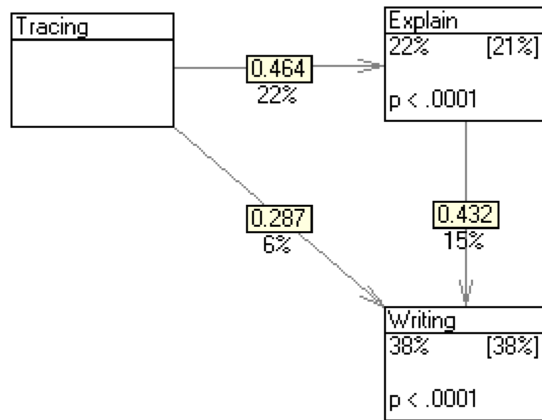

Figure 8: Path diagram for dataset PN (n=582)

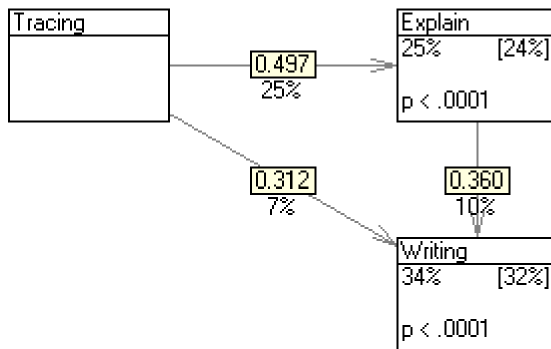

Figure 9: Path diagram for dataset PM (n=76)

[18] on two new datasets. Dataset PM was described in Section 4.3. Dataset PN comprises 582 student examination scripts from an introductory programming course for non-computing majors. Some students in the dataset had previous programming knowledge and took the course to learn Python. The course applied a procedural-first approach using Python and Eclipse. The examination, which tested both procedural and OO concepts, included three code-tracing questions, four code-explaining questions and four code-writing questions.

Our analysis of these datasets focused on the same part of the path diagram as Venables et al. [38]. A Rasch model [26] was used to convert all variables to interval level measurement and place them on a common scale ranging from 0 to 10 and centred at 5. A key feature of a Rasch model is that it is readily falsifiable and has minimal distributional assumptions, thus making it an ideal technique for relatively small datasets. We conducted a preliminary principal components analysis to verify the dimensionality of the data, and conventional Rasch quality metrics (infit and outfit) to measure the fit of the data to the model. These are essentially standardized chi-squared measures. Wright and Linacre [43] give a conventional interpretation of these, classifying items under a number of headings such as Productive, Unproductive but not degrading, and Degrading. From a measurement perspective, items that degrade measurement are deleted from the analysis; we do not expect the model to capture all the variability in the data, just the projections onto the key dimensions we are analyzing.

In dataset PN, 577 of the 582 students achieved full marks for question 1b. This identified question 1b as 'degrading' under the above measures, so that question was omitted from our analysis. All scales were uni-dimensional under principal components analysis.

In dataset PM (n=76), all questions were classified as productive except for question T15 in the writing scale which was classified as degrading. This question asked students to re-factor supplied code, which may represent a different skill set to the other code-writing questions.

Correlations between the scores in both of these independent datasets show a similar pattern to those of Lopez et al. The resulting diagrams are shown in Figures 8 and 9.

5.1 The Ordering of Tracing, Writing and Explaining

If we further study the relationship between tracing, writing and explaining in search of a possible ordering the results are more ambiguous. If a hierarchical relationship between the skills is assumed during analysis, there is some evidence that suggests tracing to be a lower-level skill than explaining or writing. However, from the data collected by the working group there are currently no results that would impose a hierarchy between the higher-level skills of explaining and writing.

It is possible that the ordering (if one exists) between explaining and writing is dependent on program size. One could speculate that explaining a program would be relatively more difficult when program size grows. While BRACElet has done a little investigation with programming assignments [25], which are generally larger than the answers to code-writing questions in examinations, the project has not yet gathered sufficient data to properly explore this question.

5.2 Exercise Construction

Another matter to consider is the coding style used in the explaining and tracing exercises. Although the code in the explaining questions is often stripped of meaningful variable and method names to discourage guessing, the coding style still follows some basic guidelines and contains little non-idiomatic code. In this sense it could be argued that the explaining task in an exam where the code resembles examples is different from a typical code comprehension task where a coder is to understand an arbitrary but meaningful piece of code.

On the other hand, tracing exercises can often include examples where the code has non-idiomatic features. Using such features disallows problem-solving strategies that more advanced students would use, forcing them to use line-by-line tracing.

The exam used by one member of the working group included a 'tricky' tracing question where the surface features of the program code, in this case meaningful strings to be printed, could lead a student to the wrong conclusion.

5.3 Discussion

From this and other BRACElet studies, we believe that the constructs we have identified and used can give us valuable insights into many of the factors that are associated with how students come to learn programming.

Table 6: Classification of annotations

Name (Code)	Description
Synchronized trace (S)	Student attempts to trace the value of more than one variable while progressing line-by-line through the code
Trace (T)	Student shows the value of a variable as it changes, showing at least one updated value
Other (O)	Student has some sort of notation that doesn't fall into either of the above categories
Blank (B)	There are no annotations.

None of these studies gives direct evidence of a hierarchy of skills. Nevertheless, they have produced reasonably consistent results across multiple institutions and programming languages. This gives us a lens through which we can evaluate potential hierarchies that are informed by theoretical considerations; such theoretical models should explain why the correlations we have found occur.

6. ANALYSIS OF DOODLES IN CODE-TRACING QUESTIONS

In this section we analyze annotations made by students on code-tracing exam questions. In an introductory analysis of such annotations, the ITiCSE Leeds Working group [15] identified 12 types of annotation, referred to as 'doodles', ranging from a blank page to a synchronized trace of variables. This work was later expanded upon by McCartney el al. [19] who compared student annotations on skeleton and fixed-code questions by quartile.

Following McCartney el al., we propose a simpler classification of annotations, a subset of those used by the Leeds group, as shown in Table 6. The differences between this classification and that of McCartney et al. are as follows. First, McCartney et al. aimed to classify all student annotations, including those used to eliminate possible answers in multiple-choice questions. In this study we aim to classify only those annotations whose perceived purpose is to help trace the given code. We therefore exclude the 'Elimination' category. Second, we maintain Synchronized Trace and Trace as separate categories. At this point we note that large variation was found in the Synchronized Trace category. Some students presented their trace in tabular format while others used less organized approaches.

We begin with an analysis of dataset PA, which is described in Section 4.2. We analyzed student annotations for three multiple-choice code-reading questions. Question PA1 involved a simple loop controlled by a compound Boolean expression. Question PA2 involved tracing a loop that processed an array. The body of the loop contained a branch and the loop was controlled by a compound Boolean expression. Question PA3 involved tracing

Table 7: Percentage of annotation types by question (dataset PA)

Question	Synch trace	Trace	Other	Blank
PA1	89.2%	1.7%	5.8%	3.3%
PA2	66.7%	12.5%	15.0%	5.8%
PA3	61.7%	14.2%	18.3%	5.8%

Table 8: Percentage of correct answers by annotation type (e.g. 86% of students with S annotations on Qn PA1 answered that question correctly)

Question	Synch trace	Trace	Other	Blank
PA1	86.0%	50.0%	57.1%	75.0%
PA2	65.0%	53.3%	50.0%	28.6%
PA3	60.8%	76.5%	31.8%	42.9%

code that contained no branch or loop but included two calls to the same function. We believe that for each of these questions, a synchronized trace would prove beneficial to students in arriving at the correct answer. We also believe that Question PA3 is more difficult than PA2, which is in turn more difficult than PA1.

Table 7 shows the percentage of annotation types observed by question. We observe that the majority of students attempt some sort of synchronized trace.

Table 8 shows the percentage of correct answers for each question by annotation type. For the first two questions we observe that students who presented a synchronized trace were indeed more likely to arrive at a correct answer. However, for Question PA3, believed to be the most complicated question, the less structured trace appeared to be more successful. This is possibly due to the fact that a synchronized trace for this question requires more sophistication and students may have had difficulty tracking local variables across multiple calls to a function.

Figure 10 shows the percentage of students who used a Synchronized Trace annotation by class quartile. In keeping with McCartney et al., we observe that the percentage of students who used Synchronized Trace annotations does not increase monotonically as we move from the lower to upper quartiles. However, we do observe that the percentage of students in the lower quartile who used a Synchronized Trace is smaller, across all questions, than in any of the other quartiles. We also note that in the lower quartiles, students are less likely to use Synchronized Trace as the level of difficulty of the question increases. Chi-squared tests show no significant correlation between Synchronized Trace and quartile for Question 1, but significant effects are found for Question 2 (effect size 8.6%, $p<0.05$) and Question 3 (effect size 12.3%, $p<0.001$).

The number of students who presented other types of trace is too small to allow for reasonable analysis.

We now examine dataset PF1. This dataset is from a second-

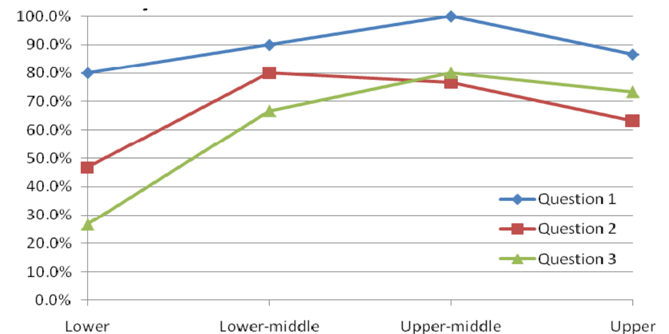

Figure 10: Percentage of students using Synchronized Trace annotations by quartile

Table 9: Percentage of annotation types by question (dataset PF1)

Question	Synch trace	Trace	Other	Blank
PF1.1	5.1%	17.9%	30.8%	46.2%
PF1.2	0.0%	0.0%	33.3%	66.7%

semester programming course of a bachelor's degree, in which the students are introduced to Perl. Most students would have studied Java the previous semester. Programming in this course consists of writing short scripts to run in a Linux environment. We have the answers of 49 students to an examination that included five code-tracing questions, five code-explaining questions, and two Parsons puzzles. We categorized the annotations presented by students with their answers to two code-reading questions. Question PF1.1 involved tracing through a loop that iterated over an array and subtracted successive elements from an initial total. Question PF1.2 asked students to determine the output from a subroutine that takes an array as a parameter and returns a Boolean to indicate whether or not the array is sorted. Table 9 shows the percentage of annotation types observed by question.

There is a noticeably lower percentage of students using Synchronized Trace or Trace than with dataset PA (Table 7), probably as a consequence of several factors. First, although there are high-level similarities between questions PF1.1-2 and PA2, the questions are certainly not the same, and do not use the same programming language. At institution PF1, students were trained to analyze code and then explain it in plain English, and questions testing this ability were posed on the examination. However, this was not tested at institution PA. It is therefore possible that, when asked to determine the output from a code segment, some students at institution PF1 approach the problem from a relational standpoint, first determining the purpose of the code and then deducing the output for the given input. This may demonstrate a more expert approach to the problem and would not require formal tracing methods such as Synchronized trace and Trace. However, this particular class consists mainly of students who did not do well in the previous semester's programming course, so it is more likely that they made no use of doodles because they have done little tracing of code.

Finally we present data gathered from institution PK. This dataset includes 93 student examination scripts from a first-semester introductory programming course in a Bachelor of Computer and Information Sciences degree. The course is taught using an objects-early approach with Java and BlueJ, and the examination addresses both procedural and OO concepts. The examination includes four code-tracing questions, four code-explaining questions, three Parsons puzzles and three writing questions.

From this dataset we categorized annotations presented by students in response to two tracing questions. Question PK1 asked students to determine the output from code that involved tracing a loop. Question PK2 involved tracing code that iterated over characters in a string and compared each character against a given character. For both questions students were asked to complete a given trace table where the variables to be traced had been identified. That is, students were explicitly led towards a Synchronized Trace. Our categorization in this case focuses only on annotations beyond those required in the given trace table. A summary is presented in Table 10.

Table 10: Percentage of annotation types by question (dataset PK)

Question	Synch trace	Trace	Other	Blank
PK1	6.5%	1.1%	46.2%	46.2%
PK2	2.2%	0.0%	58.1%	39.8%

Of particular note is that some students presented Synchronized Traces beyond those required by the question. In Question PK1, after completing the required trace table for a particular input value, students were asked to generalize the result and determine the output for a further set of input values for which trace tables were not supplied. Four of the six students who provided a Synchronized Trace in this question replicated the provided trace table to help them with this additional set of input values. The other two students created an additional table to trace the value of variables or expressions that were not included in the provided trace table.

Also of interest is that fewer than 50% of students made no additional annotations. Annotations in the Other category included noting the value of parameters above the parameter names or writing index values above individual characters in a string.

In summary, we note a large variation across institutions in the number and type of annotations provided by students. This is in keeping with the findings of McCartney et al. [19]. We also note similar variation in the style of question and the degree to which students are required to produce a formalized trace as part of their answer. As noted by McCartney et al., these differences could be due to "local culture, the way programming is taught, the way the test was administered, chance, and so forth." We also note that at one institution a Synchronized Trace was the required deliverable for some questions.

We conclude with a discussion of possible future work. We noted earlier that there was a large variation in the type of annotation that fell into the Synchronized Trace category. We recommend that future investigations attempt a finer grained classification of annotations in this category. Specifically, annotations in the form of a well organized trace table could be separated from other, less organized, formats. We also hypothesized that some students might answer tracing problems by first arriving at a relational understanding of the code. It would be interesting to investigate the correlation between success in reading code without annotation and answers to 'explain in plain English' questions that lie at the relational level on the SOLO taxonomy. We suggest that students who can explain code at the relational level may be more successful at determining the output of a particular code segment without the need to make annotations, thereby demonstrating more expert behavior.

7. SOLO ANALYSIS AND REFINEMENT

Dataset PD is from an introductory programming course taught principally to students in an Information Technology degree, using Visual Basic as the programming language. Data was collected from 97 students, a little more than half of whom have English as a second language. The exam included three 'explain in plain English' reading questions. The code in each question was quite brief, and the answers of the 91 students provided a fertile field for SOLO classification, both for examination of the

variation among raters and for teaching SOLO classification to working group members who had little or no experience in classifying answers to questions of this type.

7.1 SOLO Analysis of Reading Questions

A group of experienced classifiers worked by consensus to classify the three code-explaining questions, Q24-Q26. The member who provided this data had already concluded [31] that Q24 was a poorly chosen question, difficult both to mark and to classify. This feature arose as a consequence of that member's decision to use virtually identical pieces of code for a tracing question, a reading question, and a writing question. The tracing and writing questions were not problematic, but this particular piece of code did not readily lend itself to a satisfactory relational explanation; relational answers were typically too general to demonstrate an understanding of the code, so answers that demonstrated an understanding tended to be multistructural, that is, to be line-by-line descriptions. The SOLO classifiers agreed with this judgment, but still felt able to classify the students' answers with some confidence.

Further investigation of the SOLO levels would be facilitated by rendering them as numerical values. While there is no innate ordinality to the SOLO scale as shown in Table 1, there does appear to be a clear progression from prestructural to relational, so we replaced each nominal value with an ordinal from 0 to 5. A student's 'SOLO score' across the three questions, the sum of these values, would thus range from 0 (all answers prestructural) to 15 (all answers relational).

Figure 11 shows the number of students who scored each possible SOLO score. Visual inspection of the figure suggests a reasonably clear hump at the upper end. It is less clear whether the remainder is a single hump, two humps (split between 7 and 8) or three (further split between 3 and 4). To cover all of these eventualities, we split the SOLO scores into four equal ranges: 0 to 3, 4 to 7, 8 to 11, and 12 to 15. The groups are not quartiles – there are only 7 in the lowest group compared with 31, 26, and 27 in the others – but they nevertheless appear a reasonable division according to Figure 11.

Do the SOLO levels at which students tend to answer have any correlation with their marks on the whole exam? We divided the students into the four groups outlined above and carried out an analysis of variance to compare the means of the groups. Figure 12 shows the average mark over the whole exam of the students in each group of SOLO scores. The analysis of variance confirms that the four groups represent distinct populations (ω^2=53%; $p<0.0001$).

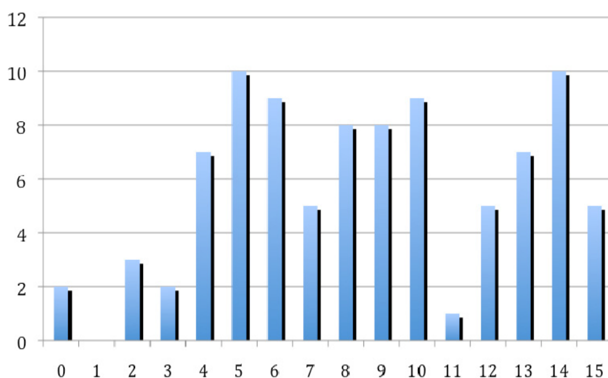

Figure 11: Number of students attaining each combined SOLO score

Inspection of the plot of means suggests a near linear relationship, so a linear regression was also performed. A Pearson correlation analysis produced a significant ($p<0.0001$) correlation of 0.76 between the SOLO score and the exam mark, accounting for 57% of the variability, slightly more than the 53% accounted for by the ANOVA. This implies that the information lost by fitting to a linear model is less than the information lost by grouping scores into categories; further, it suggests that a summated scale based on SOLO classification may be an appropriate measure for future analysis.

If the SOLO scores were marks for the code-explaining questions, or indeed for any subset of the questions on the exam, the result summarized in Figure 12 would not be at all surprising: one might reasonably expect a strong correlation between marks on some part of the exam and marks on the whole exam. But these are SOLO scores, created by assigning ordinal numbers to the SOLO categories of the students' answers, and are conceptually different from the marks for the questions. Therefore Figure 12 supports the assertion of Sheard et al. [30] that the higher the SOLO level at which students tend to explain code, the better those students tend to perform in the exam.

For further confirmation of these findings we calculated for each student a combined mark on 20 elementary multiple-choice questions, three code-tracing questions, and three comparable code-writing questions. Together these questions could be seen as the pass-level part of the exam. We created an ordinal rank for this simple exam mark (1 for the best mark to 6 for the worst mark), and for each question we conducted a Kruskal-Wallis test to compare the SOLO levels with these ranks. The tests showed excellent correlation between ranks and SOLO categories, as shown in Table 11. On Q24, the poor question, $p<0.05$, while on Q25 and Q26, $p<0.0001$. What this table means is that, with question 26, for example, students whose answers are classified as relational tended to come in the top one-sixth of the class; students whose answers are relational error tend to come in the next one-sixth, and so on, with students whose answers are prestructural tending to come in the lowest one-sixth of the class.

The same analysis was performed on the exam from dataset PM, which included five code-explaining questions. While not as clearcut as Figure 11, the histogram still displays several humps. When the students are split into three groups on the basis of the histogram, their average marks for the whole exam form a line

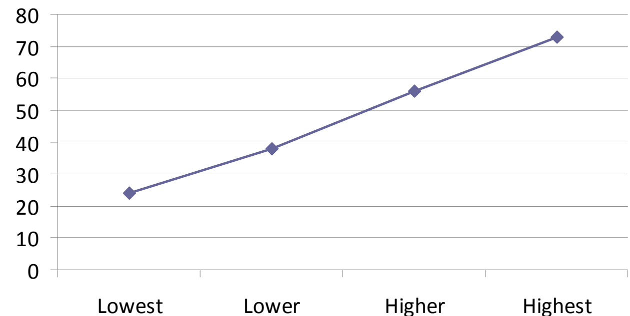

Figure 12: Average exam mark of students according to SOLO score grouping

Table 11: Ordinal simple exam ranks of students answering at each SOLO level (rank 1 indicates students who performed best on the exam)

	Q24	Q25	Q26
Relational	1	1	1
Relational Error	2	eq 2	2
Multistructural	3	eq 2	3
Multistructural Error	4	4	4
Unistructural	5	5	5
Prestructural	6	6	6

virtually parallel to that in Figure 12, but about 10% higher, suggesting that, notwithstanding local differences, the same relationship holds between students' ability to explain code and their final mark on the exam.

While there is a clear distinction between assessing students' performance and determining the level of integration in their thinking, it is pleasing to see confirmation that the higher the level of integration, the better the students tend to perform on at least the simpler component of the exam.

7.2 Reconsideration of the SOLO Categories

In addition to their classification by experienced SOLO raters, these questions were used to help teach the members of the group who had not previously used the SOLO taxonomy. The levels were explained and illustrated to the whole working group, which then jointly discussed a number of students' answers to Q26.

When it was felt that the group was reaching reasonable agreement, the SOLO novices individually classified a further set of answers, after which all of the answers were noted and discussed. There was complete agreement on only a third of the questions, and the disagreements on the remainder gave rise to serious discussion. There was a view expressed within the group that if the taxonomy cannot be applied fairly consistently by multiple raters after training that appears to be adequate, its value must be questioned. Indeed, even when the experienced raters were classifying answers by consensus there were initial disagreements, and it was clear that each position could be well argued. However, the disagreements were in the minority, and overall the extent of agreement was quite high, which would remain consistent with prior analysis [6]. As a counter to the concern over consistent allocation of SOLO ratings, it should also be noted that there was a major confound of a tendency towards generosity in marking and a student population for many of whom English was not the first language, which led to the muddying of categorizations. While grading decisions might tend to factor in equity concerns, research categorizations need to be more cut and dried.

One particular answer to one question was variously classified as unistructural, prestructural, and multistructural error, with reasonable arguments for each. Several others were classified as prestructural and unistructural, and others as unistructural and multistructural error.

We considered simply omitting from analysis any answers whose levels were not agreed upon, but it was felt that this was too dramatic. Disagreements were typically between adjacent levels, and omitting such answers would mean discarding potentially valuable data. Perhaps there was a way of collapsing the levels that gave rise to most of the disagreement. We are interested in whether students can produce a correct and complete explanation, and in whether that explanation is relational or multistructural. Answers in the three lowest levels (multistructural error, unistructural, and prestructural) all tend to indicate that the students have not displayed a real understanding of the code, and perhaps it is not worth a great deal of time and effort distinguishing between these levels. A simple solution would be to collapse the lowest three levels of the taxonomy into a single 'Other' category.

Table 12: Proposed reduced SOLO taxonomy for code-explaining answers

SOLO category	Description
Relational (R)	A summary of what the code does in terms of its purpose (the forest)
Relational Error (RE)	A summary of what the code does in terms of its purpose, but with some minor error
Multistructural (M)	A line-by-line description of all the code (the trees)
Other (O)	Any other description of part or all of the code, displaying no real evidence of understanding of the code as a whole

Of more interest were a number of questions that attracted ratings of relational error, a fairly high rating, and prestructural, the lowest rating. A model answer to Q26 is "It finds the longest title (in an array of titles)." An answer such as "It sorts the titles," is clearly a relational answer; but is it a guess, which should be rated prestructural, or an informed but wrong conclusion, which should be rated relational error? In the answer "It finds the longest title and puts it in the first position," is the second part a major error, suggesting a prestructural rating, or a minor one, permitting a rating of relational error? In essence, does the answer show major or minor misunderstanding of the purpose of the code?

The solution to this problem was not to collapse the relational error category, which does appear to sit clearly between the (correct) relational and the (correct) multistructural, but to be more prescriptive about when to apply it. Specifically, a relational answer that is all but correct, but showing a minor error such as the confusion of > with <, will be classified as relational error. Other answers that are relational in form but not essentially correct will be classified as prestructural, or as other if the collapsed taxonomy is being used. So "It finds the shortest title" would be classified relational error, as would "It finds the longest title and puts it in the first place"; whereas "It sorts the titles" would be classified as prestructural or other.

In summary, the workshop proposes a further iteration of the SOLO taxonomy for the classification of code-reading questions, as shown in Table 12. General observations by the team indicated considerably higher levels of consensus on this reduced set of categories, and future work will include a formal inter-rater reliability analysis of the categorizations.

8. APPLYING THE SOLO TAXONOMY TO CODE-WRITING QUESTIONS

While it is hoped that examination marking schemes give some measure of a student's acquisition of programming skills, they do not necessarily indicate the type of skill displayed by a student or the level at which that skill is displayed. For this reason the BRACElet project has developed and used a set of SOLO categories for code-reading (explain in English) questions, as addressed in several of the preceding sections.

On the other hand, little work has been done on a comparable system for categorizing students' answers to code-writing questions. Analysis of answers to code-writing questions [16, 18] has instead relied upon the examiners' marking strategies, whose purpose is quite different from that of our research.

8.1 Developing Categories

How might the SOLO taxonomy be adapted for code-writing exercises, and what types of question might be usefully examined with such a taxonomy? Biggs [2] describes the types of verb that apply for each of the levels of the taxonomy (p47) and provides an example of ordering outcome items by the taxonomy (pp176-178). He later provides an example of how SOLO can be applied to assessing portfolios (pp213-221). Thompson [35, 36] adapts Biggs's portfolio example for the assessing of programming assignments and essays.

Hattie and Purdie [13 p156] provide a number of examples of the use of the SOLO taxonomy. The example for language translation, as shown in Table 13, provides some insights for the application of SOLO to writing programs as it explains the different levels of translation of the French phrase 'sa table de nuit'.

The shift through the SOLO levels shows an increasing understanding of how the phrase should be interpreted rather than just translated. It shows an increasing awareness of the relationship between the words and how that relationship communicates meaning. The unistructural translation is a word-for-word translation. Although it carries the meaning, it doesn't portray the underlying intent of the original phrase. The multistructural translation recognizes that "of the" is a poor translation and that "at" better describes the meaning of the original phrase (that is, the table that he uses at night) but it is still effectively little more than a word-for-word translation. At the relational level there is a recognition that the English word order can be revised, providing an improved form of the phrase with a clearer meaning but without an interpretation into the context of English. Finally, at the extended abstract level the context of the table's usage is drawn into the translation, thus portraying the real nature of the table – at least to a speaker of British English. Hattie and Purdie appear unaware that in American English this piece of furniture is generally called a night table or a nightstand.

In attempting to apply this thinking to the programming context, the working group proposed the categories shown in Table 14.

Table 13: SOLO taxonomy in language translation (from Hattie & Purdie [13])

Translate:	'sa table de nuit'
Unistructural	His table of the night
Multistructural	His table at night
Relational	His night table
Extended abstract	His bedside table

For each of the categories above Prestructural, modifiers were introduced to indicate errors in the student's solution. These modifiers are:

- e – generally satisfies the requirements of the level but has minor syntax or logic errors.
- E – generally satisfies the requirements for the level but has significant syntax or logic errors.

8.2 Analysis

The proposed levels in Table 14 were initially developed by attempting to apply SOLO principles to a code-writing question from dataset PN that required the writing of three conditional statements. For this initial problem, a direct translation produced a working solution in which the sequence of the conditional statements was irrelevant. Removing redundant conditions and utilizing else clauses moved the solution toward a relational solution. This exercise was performed as a theory exercise without analyzing student responses to the question. The solutions proposed were all valid program segments that satisfied the specifications. The question structure appeared to encourage a *direct translation* (unistructural response) or a *refinement* (multistructural [M] response) but didn't rule out the possibility of an *encompassing* (relational) or *extending* (extended abstract) response. We have therefore applied a classification of M to the question itself. Classifying questions in this way is a recognition that one can hardly expect students to produce relational or extended abstract answers to questions structured so as to invite multistructural or unistructural answers.

To test the proposed categories we analyzed students' answers to three writing questions. From dataset PK we classified 59 answers to a complex question related to the sale of theatre tickets; from dataset PN, 30 answers to a question involving three conditional statements; and from dataset PF1, 30 answers to a question requiring calculation of an average from a set of input data. In this last question there was an incentive for the students to use a

Table 14: SOLO categories for code-writing tasks

Phase	SOLO category	Description
Qualitative	Extended Abstract – Extending [EA]	Uses constructs and concepts beyond those required in the exercise to provide an improved solution
	Relational – Encompassing [R]	Provides a valid well structured program that removes all redundancy and has a clear logical structure. The specifications have been integrated to form a logical whole.
Quantitative	Multistructural – Refinement [M]	Represents a translation that is close to a direct translation. The code may have been reordered to make a valid solution.
	Unistructural – Direct Translation [U]	Represents a direct translation of the specifications. The code will be in the sequence of the specifications.
	Prestructural [P]	Substantially lacks knowledge of programming constructs or is unrelated to the question.

Table 15: SOLO categorizations for code-writing datasets

Dataset	PK	PN	PF1
N	59	30	30
Question level	R	M	M+
EA		1	
R	8	2	13
Re	14		
RE	5		
M		6	12
Me			
ME	3	1	1
U		17	1
UE	16	3	
P	13		3

subroutine, so we have classified this question level as M+ rather than M. Our analysis of the three questions provided the distribution shown in Table 15.

The nature of the question for the sale of tickets (PK) promoted an encompassing (relational) response, so we classified the question level as R. A valid solution to the question required the integration of all of the individual specifications. While several responses were at encompassing (Relational) level, it was notable that a large proportion had either small or large errors. A direct translation (Unistructural) or refined translation (Multistructural) might produce a program that would compile but would not execute in a way that would produce the desired result. This is reflected in the pattern of error codes for these two categories.

For the dataset PN question involving three conditional statements, the data confirmed the initial conjecture that patterns would be within a restricted range, with direct translation (Unistructural) responses predominating and few responses in the higher categories.

For the larger dataset (PK) a non-parametric Kruskal-Wallis test was conducted to test the ordinality of the categories. The large and small error sub-categories were collapsed. Significant rankings were identified ($p=0.0001$), consistent with the notion that the SOLO scales were progressive and represented valid ordinal categories.

This tabulation supports the view that this SOLO taxonomy for writing represents an ordinal scale, suggesting that the definitions used for categorization are reasonable for this analysis. These findings appear consistent with the earlier conclusions relating to the SOLO reading scales (Section 7.1). The results achieved to date on this limited set of questions, and the coherence of the resulting patterns of responses, suggest that this is a promising way of categorizing performance on tasks involving the writing of program code. It is also evident from this limited study that the level at which the question is set may constrain the natural range of responses from a SOLO perspective.

8.3 Progress in Applying SOLO to Code Writing

The analysis of Section 8.2, which was performed by three members of the working group, suggests that the proposed categories may be appropriate for the analysis of code-writing questions. It appears that the process of defining SOLO ratings for a specific question needs to be interpreted based upon the level and nature of the question and the patterns that emerge from the students' answers. Further analysis against different code-writing exercises is required to investigate the comprehensiveness of the scheme and to confirm the validity of the categories. The degree to which they can be reliably applied by multiple raters (as, for example, in Clear et al. [6] for code-reading questions) will also require further study.

With the analysis of code-explaining questions, certain types of code segment provide improved opportunities for performing SOLO analysis. This would appear to hold equally true for code-writing exercises. Code-writing questions aimed at specific SOLO levels could provide a mechanism to assess the level of performance of students in code writing. For example, 'Write a statement that assigns the value 2 to the variable x' is aimed at a unistructural (direct translation) response, and would assess the students' performance at this elementary level.

Of more importance to the BRACElet studies is the potential for this form of analysis to complement our analysis of reading tasks and develop stronger understandings of the relationships between reading and writing. It is hoped that in due course the insights gained from this type of analysis will help in theory building and the development of strong predictive models that depict how knowledge is developed in the introductory programming process.

Apart from the improvement of understanding about the learning process, we believe that this work will provide deep insights into more sophisticated teaching and assessment strategies for introductory programming.

9. FUTURE WORK

This working group was intended to broaden and deepen the repository of data available to the BRACElet project, to use this enhanced data set to support or question earlier findings, and to broaden the theoretical basis for the work. At the same time, though, it was intended to draw further participants to this area of study. This section indicates some of the tasks that might be considered both by current members of the project and by anyone else who is interested in pursuing this line of research.

9.1 Replication

A number of the findings of the BRACElet project, both prior to and at this working group, are interesting but unconfirmed. The replication studies conducted within this report have served to confirm some definite patterns, but there is still more to be done. There is a clear place for further replication of the work.

9.2 Is there a Hierarchy?

Does novice programmers' acquisition of programming skills form a hierarchy, as our work suggests? Indeed, what do we mean by hierarchy? What sort of evidence would support our current beliefs about a hierarchy? More important, what sort of evidence would falsify a proposed hierarchy?

Assuming the existence of a hierarchy, it is unlikely that we have discovered all of its useful levels. What other levels might there be, and what sort of question would test students at those levels? One other level might involve the ability to turn detailed diagrams into code, to turn code into diagrams, and to match corresponding diagrams and code.

As Figure 1 suggests, there are successive layers of conceptual understanding to be navigated in developing programming skills. It has been reported that it takes some ten years for a programmer to become proficient [27], in a process that would naturally involve successive levels of skill development. Since the scope of this BRACElet study relates mainly to a hierarchy for novice programmers, what might the higher levels look like, and how might they be discerned? Philpott et al. [25], investigating the performance of programmers at a more intermediate level, provide an early view of how those skills might be investigated.

9.3 Fish with Legs

The evolutionary record is well populated with members of stable stages, while evidence of intermediate developmental stages is rare or non-existent. If there were a hierarchy of programming skills, it would almost certainly form a continuum rather than a series of discrete levels. If this is so, what might constitute evidence for transitional stages between the currently recognized levels?

9.4 Pedagogy

If there is a hierarchy of programming skills, it would be productive to investigate possible pedagogical ramifications of this finding. How might the knowledge of this hierarchy inform our teaching? How might students benefit from an acceptance of the hierarchy? How might we teach students to see the hierarchy?

Studies of the roles of variables [4] might prove fruitful in this regard, as might other approaches we have not yet considered.

9.5 Think-aloud Studies

Some of the questions that we have studied, notably the explain-in-plain-language questions, appear to suffer from our inability to clearly indicate to the students exactly what level of integration we require in the answers [31]. It is easy for students to give answers that are too detailed; but as they learn not to do this, it is equally easy for them to give answers that are so general that they fail to indicate a full understanding of the code. It might be worth considering setting these questions not as exam questions but as think-aloud questions, in which we can explore the participants' understanding through their descriptions of the process, and indeed through follow-up questions if we remain unclear as to the level of their understanding. Think-aloud questions have already been used by BRACElet, not with novice students but with expert programmers [17].

9.6 New Types of Question

A number of new types of exam question merit consideration both as assessment items and as items for exploration in our research. They include:

- Content questions: an indication (either in words or shown diagrammatically) of what code is required to do, followed by question such as whether the code will require a loop.
- Debugging questions: explain what a given piece of code is supposed to do, and point out that has an error on one line; ask students to identify the line and correct the error.
- Modifying questions: ask students to modify a piece of code so that it instead of doing one task it does another. (A colleague of one of the working group leaders began such a question by asking students to write the code for the first task; students struggled with this question.)
- Syntactic error recognition questions, perhaps in the style of Wiedenbeck [39].
- Style comprehension questions, as in the same study by Wiedenbeck [39].
- Questions presenting a number of pieces of code and asking which pieces achieve the same outcome.
- A card sort experiment using pieces of code.

9.7 Students beyond the First Course

The bulk of the data analyzed by BRACElet has been from introductory programming courses. Will further study of students in advanced courses show the same characteristics, will it show analogous characteristics, or will it suggest that our findings have no bearing on students at more advanced stages of their degrees?

9.8 Other Types of Code Fragment

BRACElet has to date concentrated on short loops with or without selection logic inside the loop. There are clearly other programming constructs that merit study, and work should be carried out on these. In addition, none of the BRACElet work to date (with the exception of Philpott et al. [25]) has explored students' skills in object-oriented programming, functional programming, or other programming paradigms.

9.9 Study with a Control Group

It is a possible weakness of our work that we do not conduct controlled experiments. Indeed, it is unlikely to be ethically acceptable to do this in the exam context. In other contexts, though, such as think-aloud questions, we should consider, perhaps, a 'trick' question with a non-trick variant for a control group. We would then measure how the more successful and less successful students perform on the question.

10. DISCUSSION AND CONCLUSIONS

This report paints a brief sketch of research that can be done in analyzing the naturally occurring data of students' answers to examination questions. The findings presented here should be seen as indicative, not as limiting. Essentially, we believe that any research into what and how students learn can be enhanced by studying their assessment; and that this form of study is considerably easier in many respects than studies with purpose-designed data collection.

Of course there are limitations to the research that can be done on examination answers. Researchers must remain constantly vigilant to ensure that the questions they set are driven by the primary purpose of examinations, the assessment of students. While an examination question might be written because of the potential research value of its answers, it must nevertheless be a valid assessment item for the course in question. If there is ever any doubt about this, the question should not be used.

Another limitation, already discussed, is the size of programs that students can reasonably be expected to trace, read, or write in an examination. We as educators might like to believe that programming skills demonstrated on short pieces of code are indicative of comparable skills with much larger programs, but this scalability has never been established, so we must remain aware that conclusions reached in working with short code segments really apply only to short code segments rather than to programming as a whole.

Despite these limitations, the analysis of students' examination answers offers great potential for research into what and how students are learning.

The specific work presented in this paper, the product of an ITiCSE working group, is very much a work in progress. Beginning with the findings of the Leeds ITiCSE working group in 2004 [15], the BRACElet project has begun to gather and develop tools and theories for exploring the development of programming skills in novice students. Located in the context of that broader study, this paper presents some findings from the work of subgroups addressing different aspects of the theory, the instruments, and empirical analysis of a large body of data (some 1300 student scripts across seven different institutional contexts).

Several new contributions to the BRACElet work are presented here. A refined set of SOLO taxonomies for categorization of reading questions has been defined. For the first time, a SOLO taxonomy for writing questions is presented, based upon theoretical analysis and refined through its application to three empirical datasets. This latter taxonomy is very much in its infancy, and appears to be quite context-specific in its application. Both taxonomies do appear to be ordinal, which is an encouraging finding, and there does at this stage appear to be a clear distinction between the two.

The links between the BRACElet work and theoretical work from mathematics education, and between that work and the refined SOLO taxonomies, appear to offer a promising theoretical basis for investigating a graduated hierarchy of skill development in programming.

This report includes several replication studies of prior work, with outcomes generally consistent with the earlier studies. The larger and more diverse datasets to which the working group has had access lend strong support to the patterns of hierarchy and potential hierarchy that have been noted. Work now remains to develop more specific hypotheses based upon these findings and build stronger theories capable of being readily refuted, and by implication capable of supporting conclusions.

The individual findings presented here are simple overviews of what was actually found, but they help to consolidate a growing set of understandings about the ways in which novice programmers develop expertise. We expect that far more will be found as we delve more deeply into the rich set of data that was brought to the working group. For instance we see considerable scope for linking the empirical findings to more theoretical understandings of the development of programming expertise, and hope that we can use this data to help confirm or refute the more specific sets of hypotheses that we are now developing.

Finally, we encourage others, whether or not they are currently involved with BRACElet, to gather their own data (perhaps guided by the BRACElet 2009.1 (Wellington) specification), to conduct their own analysis (perhaps guided by the analysis presented here), and to add their own contributions to this promising field of study.

11. ACKNOWLEDGEMENTS

The working group members wish to thank Kerttu-Pollari Malmi for her work in support of the group and her kind cooperation in allowing us to collect data from her course. We also acknowledge the prior contributions of Jacqueline Whalley as one of the principal investigators in the BRACELet project, although she was unfortunately unable to participate in this working group. The generous past financial support of AUT University, ACM SIGCSE and the Australian Learning and Teaching Council for the BRACElet project is also acknowledged. Finally, we are grateful to the reviewers of this report, whose suggestions have unquestionably made it a better paper.

12. REFERENCES

[1] Baroody, A., Feil, Y., and Johnson, A. (2007). An alternative reconceptualization of procedural and conceptual knowledge. *Journal for Research in Mathematics Education*, 38:2, 115-131.

[2] Biggs, J. B. (1999). *Teaching for quality learning at university*. Buckingham: Open University Press.

[3] Biggs, J. B. and Collis, K. F. (1982). Evaluating the quality of learning: the SOLO taxonomy (Structure of the Observed Learning Outcome). New York: Academic Press.

[4] Byckling, P. and Sajaniemi, J. (2006). Roles of variables and programming skills improvement. *SIGCSE Bulletin* 38:1, 413-417.

[5] Clear, T., Edwards, J., Lister, R., Simon, B., Thompson, E. and Whalley, J. (2008). The teaching of novice computer programmers: bringing the scholarly-research approach to Australia. *Tenth Australasian Computing Education Conference (ACE 2008)*, Wollongong, Australia, 63-68.

[6] Clear, T., Whalley, J., Lister, R., Carbone, A., Hu, M., Sheard, J., Simon, B. and Thompson, E. (2008). Reliably classifying novice programmer exam results using the SOLO taxonomy. *21st Annual NACCQ Conference*, NACCQ, Auckland, New Zealand, 23-30.

[7] Colburn, T. and Shute, G. (2007). Abstraction in computer science. *Minds & Machines* 17, 169-184.

[8] Cottrill, J., Dubinsky, E., Nichols, D., Schwingendorf, K., Thomas, K., and Vidakovic, D. (1996). Understanding the limit concept: beginning with a coordinated process scheme. *Journal of Mathematical Behavior* 15:2, 167-192.

[9] Denny, P., Luxton-Reilly, A. and Simon, B. (2008). Evaluating a new exam question: Parsons problems. *Fourth International Workshop on Computing Education Research (ICER '08)*, Sydney, Australia, 113-124.

[10] Fincher, S., Lister, R., Clear, T., Robins, A., Tenenberg, J. and Petre, M. (2005). Multi-institutional, multi-national studies in CSEd research: some design considerations and trade-offs. in Anderson, R., Fincher, S. and Guzdial, M. eds. *First International Workshop on Computing Education Research*, Seattle, WA, USA, 111-121.

[11] Gray, E., Pinto, M., Pitta, D., and Tall, D. (1999). Knowledge construction and diverging thinking in elementary and advanced mathematics. *Educational Studies in Mathematics* 38, 111-133.

[12] Gray, E. and Tall, D. (2007). Abstraction as a natural process of mental compression. *Mathematics Education Research Journal* 19:2, 23-40.

[13] Hattie, J. and Purdie, N. (1998). The SOLO model: addressing fundamental measurement issues. In Dart, B. & Boulton-Lewis, G. M. (Eds.), *Teaching and learning in higher education*, 145-176. Camberwell, Vic: Australian Council of Educational Research.

[14] Hiebert, J. and Lefevre, P. (1986). Conceptual and procedural knowledge in mathematics: an introductory analysis. In Hiebert, J., ed., *Conceptual and procedural knowledge: the case of mathematics*, 1-27. Erlbaum, Hillsdale, NJ.

[15] Lister R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin* 36:4, 119-150.

[16] Lister, R., Fidge, C., and Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *14th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'09)*, July 3-8, 2009, Paris, France.

[17] Lister, R., Simon, B., Thompson, E., Whalley, J. L., and Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *SIGCSE Bulletin* 38:3, 118-122.

[18] Lopez, M., Whalley, J., Robbins, P., and Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. *Fourth International Workshop on Computing Education Research (ICER '08)*, Sydney, Australia, 101-112.

[19] McCartney, R., Mostrom, J.E., Sanders, K., and Seppälä, O. (2004). Questions, annotations, and institutions: observations from a study of novice programmers. *Fourth Finnish / Baltic Sea Conference on Computer Science Education (Koli Calling 2004)*, 11-19.

[20] McCormick, R. (1997). Conceptual and procedural knowledge. *International Journal of Technology and Design Education* 7:1-2, 141-159.

[21] Parsons, D. and Haden, P. (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. *Eighth Australasian Computing Education Conference (ACE2006)*, Hobart, Australia, 157-163.

[22] Pegg, J. (2002). Assessment in mathematics: A developmental approach. In J. M. Royer (Ed.), *Mathematical Cognition* 227-259. Information Age Publishing: Greenwich, CT, USA.

[23] Pegg, J. and Tall, D. (2005). The fundamental cycle of concept construction underlying various theoretical frameworks. *ZDM* 37:6, 468-474.

[24] Perkins, D. and Martin, F. (1989). Fragile knowledge and neglected strategies in novice programmers. In Soloway, E. and Spohrer, J, Eds., *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, NJ, USA, 213-229.

[25] Philpott, A., Robbins, P., and Whalley, J. (2007). Accessing the steps on the road to relational thinking. 20th Annual Conference of the National Advisory Committee on Computing Qualifications, Nelson, New Zealand, 286.

[26] Rasch, G. (1960/1980). *Probabilistic models for some intelligence and attainment tests* (Copenhagen, Danish Institute for Educational Research), expanded edition (1980) with foreword and afterword by B.D. Wright. Chicago: The University of Chicago Press.

[27] Robins, A., Rountree, J., and Rountree, N. (2003). Learning and teaching programming: a review and discussion. *Computer Science Education* 13:2, 137- 172.

[28] Sfard, A. (1988). Operational vs. structural method of teaching mathematics – case study. Twelfth Conference for the Psychology of Mathematics Education 2, 560-567. Vesprèm, Hungary: Ferenc Genzwein, OOK.

[29] Sfard, A. (1991). On the dual nature of mathematical conceptions: reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics* 22, 1-36.

[30] Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., and Whalley, J. L. (2008). Going SOLO to assess novice programmers. *SIGCSE Bulletin* 40:3, 209-213.

[31] Simon (2009). Code-explaining exam questions: a cautionary note. *Ninth International Conference on Computing Education Research (Koli Calling 2009).*

[32] Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM* 29:9, 850-858.

[33] Star, J. R. (2005). Reconceptualizing procedural knowledge. *Journal for Research in Mathematics Education* 36, 401-411.

[34] Tall, D., (2008). The transition to formal thinking in mathematics. *Mathematics Education Research Journal* 20:2, 5-24.

[35] Thompson, E. (2004). Does the sum of the parts equal the whole? *21st Annual NACCQ Conference*, NACCQ, Auckland, New Zealand, 2008, 440-445.

[36] Thompson, E. (2007). Holistic assessment criteria - Applying SOLO to programming projects. *Ninth Australasian Computing Education Conference (ACE2007)*, Ballarat, Australia, 155-162.

[37] Thompson, E. (2009). *How do they understand? Practitioner perceptions of an object-oriented program*. Dissertation, Massey University, Palmerston North.

[38] Venables, A., Tan, G., and Lister, R. (2009) A closer look at tracing, explaining and code writing skills in the novice

programmer. *Fifth International Workshop on Computing Education Research (ICER '09)*, Berkeley, California, USA.

[39] Wiedenbeck, S. (1985). Novice/expert differences in programming skills. *International Journal of Man-Machine Studies* 23:4, 383-390.

[40] Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A., and Prasad, C. (2006). An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. *Eighth Australasian Computing Education Conference (ACE2006)*, Hobart, Australia, 243-252.

[41] Whalley, J. and Lister, R. (2009). The BRACElet 2009.1 (Wellington) Specification. *Eleventh Australasian Computing Education Conference (ACE2009)*, Wellington, New Zealand, 9-18.

[42] Whalley, J. and Robbins, P. (2007). Report on the Fourth BRACElet Workshop. *Bulletin of Applied Computing and IT*. Retrieved June 7, 2007 from http://www.naccq.co.nz/bacit/0501/2007Whalley_BRACELET_Workshop.htm.

[43] Wright, B.D, and Linacre, J.M. (1994) Reasonable mean-square fit values. *Rasch Measurement Transactions* 8:3, 370.