Conferences in Research and Practice in
Information Technology

Volume 95

Computing Education 2009

Australian Computer Science Communications, Volume 31, Number 5

AUSTRALIAN
COMPUTER
SOCIETY

acsw

COmputing
Research
& Education

Computing Education 2009

Proceedings of the
Eleventh Australasian Computing Education Conference
(ACE 2009), Wellington, New Zealand,
January 2009

Margaret Hamilton and Tony Clear, Eds.

Volume 95 in the Conferences in Research and Practice in Information Technology Series.
Published by the Australian Computer Society Inc.

Published in association with the ACM Digital Library.

Computing Education 2009. Proceedings of the Eleventh Australasian Computing Education Conference (ACE 2009), Wellington, New Zealand, January 2009

Conferences in Research and Practice in Information Technology, Volume 95.

Copyright ©2009, Australian Computer Society. Reproduction for academic, not-for-profit purposes permitted provided the copyright text at the foot of the first page of each paper is included.

Editors:

Margaret Hamilton
School of Computer Science and Information Technology
RMIT University
GPO Box 2476V
Melbourne, Victoria, 3001, Australia
Email: `margaret.hamilton@rmit.edu.au`

Tony Clear
School of Computing and Mathematical Sciences
Auckland University of Technology
Private Bag 92006
Auckland 1020, New Zealand
Email: `tony.clear@aut.ac.nz`

Series Editors:
Vladimir Estivill-Castro, Griffith University, Queensland
John F. Roddick, Flinders University, South Australia
Simeon Simoff, University of Western Sydney, NSW
`crpit@infoeng.flinders.edu.au`

Publisher: Australian Computer Society Inc.
PO Box Q534, QVB Post Office
Sydney 1230
New South Wales
Australia.

Conferences in Research and Practice in Information Technology, Volume 95.
ISSN 1445-1336.
ISBN 978-1-920682-76-7.

Printed, January 2009 by Flinders Press, PO Box 2100, Bedford Park, SA 5042, South Australia.
Cover Design by Modern Planet Design, (08) 8340 1361.

The *Conferences in Research and Practice in Information Technology* series aims to disseminate the results of peer-reviewed research in all areas of Information Technology. Further details can be found at http://crpit.com/.

Table of Contents

Preface

Welcome to the Eleventh Australasian Computing Education Conference (ACE2009). This year, the ACE2009 conference, which is part of the Australasian Computer Science Week, is being held in Wellington, New Zealand from January 19 to January 23, 2009.

We can truly call this an international conference with 77 authors coming from Malaysia, Taiwan, China, Finland, England, United States, Greece, Argentina, Sweden, New Zealand and Australia. The Chairs would like to thank the Program Committee for their excellent efforts in the double-blind reviewing process which resulted in the selection of 18 full papers from the 40 papers submitted, giving an acceptance rate of 45%.

Our keynote speaker is Professor Mark Guzdial from Georgia Tech, author of several books including ”Introduction to Computing and Programming with Python: A Multimedia Approach.” He is currently vice-chair of the ACM Education Board and is a prominent member of SIGCSE, being the Symposium Co-Chair for SIGCSE 2009. For two days prior to our conference, we have organized a workshop partly sponsored by SIGCSE on ”Contextualised Approaches to Computing Education”. The presenters, Mark Guzdial and his wife Barbara Ericson, will cover several different contextualized approaches, including media computation, robotics and engineering approaches to CS1.

The topics of ACE2009 papers and presentations include taxonomies, classifications, studies of novice programming students, the use of technology in education, course content, curriculum structure, methods of assessment, mobile, flexible, online learning, and evaluations of alternative approaches to computing education. The high quality papers this year continue to push the frontiers of opportunities for research and innovation in computing education, and this conference will enable these educators to meet and share their experiences in a new forum. We will be holding a Second Life Panel where we will attempt to stream SL to the Conference room and connect Melbourne, Auckland and Nelson to showcase SL in action.

In keeping with the ACE tradition, there will be a post-conference workshop continuing to build research in Australasian computing education. Five years ago in Dunedin, New Zealand, we held the first BRACE workshop, and this year, on the return to New Zealand we are continuing the tradition by holding a BRACElet workshop.

We are grateful to SIGCSE for the grant to fund the pre-conference workshop, and for sponsoring the Conference jointly with the ACM. We thank everyone involved in Australasian Computer Science Week for making this Conference and Proceedings publication possible, and we thank CORE, our hosts Victoria University Wellington, and the Australasian Computing Education Executive for the opportunity to chair this ACE2009 Conference.

Margaret Hamilton
RMIT University, Australia

Tony Clear
Auckland University of Technology, New Zealand

ACE 2009 Programme Chairs
January 2009

Programme Committee

Chairs

Margaret Hamilton, RMIT University (Australia) (senior co-chair)
Tony Clear, Auckland University of Technology (New Zealand) (co-chair)

Members

Alison Young, Unitec (New Zealand)
Angela Carbone, Monash University (Australia)
John Hamer, Auckland University (New Zealand)
Judy Kay, University of Sydney (Australia)
Judy Sheard, Monash University (Australia)
Mats Daniels, Uppsala University (Sweden)
Michael de Raadt, University of Southern Queensland (Australia)
Raymond Lister, University of Technology Sydney (Australia)
Sally Fincher, University of Kent (UK)
Simon, University of Newcastle (Australia) (senior co-chair)

Additional Reviewers

A. John Hurst, Monash University (Australia)
Alan Fekete, University of Sydney (Australia)
Alanah Kazlauskas, Australian Catholic University (Australia)
Arnold Pears, Uppsala University (Sweden)
Daryl D'Souza, RMIT University (Australia)
Dave Bremer, Otago Polytechnic (New Zealand)
Logan Muller, Unitec (New Zealand)
Martin Dick, RMIT University (Australia)

Organising Committee

Co-Chairs

Dr Alex Potanin
Professor John Hine

Venues

Dr David Pearce

Operations

Dr Peter Komisarczuk
Mrs Suzan Hall
Mr Craig Anslow

Finance and Program

Dr Stuart Marshall

Communications

Dr Ian Welch
Mr Craig Anslow

Events

Professor John Hine

Welcome from the Organising Committee

We would like to welcome you to ACSW2009 hosted by Victoria University of Wellington, New Zealand.

Wellington is set on the edge of a stunning harbour and surrounded by rolling hills. The earliest name for Wellington, from Maori legend, is Te Upoko o te Ika a Maui. In Maori it means the head of Maui's fish. Caught and pulled to the surface by the Polynesian navigator Maui, the fish became the North Island. Wellington is the capital city of New Zealand and home to the seat of parliament. But this vibrant and dynamic city also has many other capital claims including Culture capital, Creative capital and Events capital. It is a compact, walkable city waiting to be explored. The conference venue is less than fifteen minutes walk to accommodation, Courtenay Place with its wide range of bars, and the harbour with its restaurants and activities such as sea kayaking. The conference venue itself is in the Museum of New Zealand Te Papa Tongarewa, offering visitors a unique and authentic experience of this country's treasures and stories. Over five floors, you can explore the nation's nature, art, history, and heritage - from the shaping of its land to the spirit of its diverse peoples, from its unique wildlife to its distinctive art and visual culture.

Victoria University of Wellington - Te Whare Wānanga o te Ūpoko o te Ika a Māui - is over a century old. Victoria College was founded through an Act of Parliament in 1897, the year of Queen Victoria's Diamond Jubilee celebrations, and named in her honour. Victoria is a thriving community of almost 25,000 people. Situated in the capital city across four campuses, Victoria can take advantage of connections and values its relationships with iwi, business, government, the judiciary, public and private research organisations, cultural organisations and resources, other universities and tertiary providers and the international community through the diplomatic corps. ACSW2009 coincides with the opening of the new School of Engineering and Computer Science as part of the Faculty of Engineering at Victoria University of Wellington - combining a long history of research and teaching of the software engineering and network engineering in the Computer Science department and computer system engineering and electronic engineering in the Physics department. Professor John Hine, co-chairing ACSW2009, is the current Dean of Engineering and the inaugural Head of School of Engineering and Computer Science.

ACSW2009 consists of the following conferences:

– Australasian Computer Science Conference (ACSC) (Chaired by Bernard Mans),
– Australasian Computing Education Conference (ACE) (Chaired by Margaret Hamilton and Tony Clear),
– Australasian Database Conference (ADC) (Chaired by Athman Bouguettaya and Xuemin Lin),
– Australasian Symposium on Grid Computing and e-Research (AUSGRID) (Chaired by Wayne Kelly and Paul Roe),
– Computing: The Australasian Theory Symposium (CATS) (Chaired by Prabhu Manyem and Rod Downey),
– Asia-Pacific Conference on Conceptual Modelling (APCCM) (Chaired by Markus Kirchberg and Sebastian Link),
– Australasian Information Security Conference (AISC) (Chaired by Ljiljana Brankovic and Willy Susilo),
– Australasian Workshop on Health Informatics and Knowledge Management (HIKM) (Chaired by Jim Warren),
– Australasian User Interface Conference (AUIC) (Chaired by Gerald Weber and Paul Calder),
– Australasian Computing Doctoral Consortium (ACDC) (Chaired by David Pearce and Vladimir Estivill-Castro).

The nature of ACSW requires the co-operation of numerous people. We would like to thank all those who have worked to ensure the success of ACSW2009 including the Organising Committee, the Conference Chairs and Programme Committees, our sponsors, the keynote speakers and the delegates.

Dr Alex Potanin and Professor John Hine
ACSW2009 Co-Chairs
Victoria University of Wellington
January, 2009

CORE - Computing Research & Education

CORE welcomes all delegates to ACSW2009 in Wellington. CORE, the peak body representing academic computer science in Australia and New Zealand, is responsible for the annual ACSW series of meetings, which are a unique opportunity for our community to network and to discuss research and topics of mutual interest. The original component conferences – ACSC, ADC, and CATS, which formed the basis of ACSW in the mid 1990s – now share the week with seven other events, which build on the diversity of the Australasian CS community.

This year, we have chosen to feature a small number of plenary speakers chosen from across the discipline, Ronald Fagin, Ian Foster, Mark Guzdial, and Andy Hopper. I thank them for their contributions to ACSW'09.The efforts of the conference chairs and their program committees have led to strong programs in all the conferences – again, thanks. And thanks are particularly due to Alex Potanin, John Hine, and their colleagues for organising what promises to be a memorable ACSW.

In Australia, 2008 has been a busy year for academia, with the incoming Labor government instituting major reviews in areas such as the higher education sector, research funding, postgraduate study, and national curricula. However, while the reviews have exposed severe shortcomings in the funding of higher education and research, they have not as yet been translated into definite action, and the sector as a whole is shrinking. Although there is a widespread perception of a shortage of IT staff, and graduate salaries remain strong, student interest in ICT continues to be low. Moreover, per-place funding for computer science students has dropped relative to that of other physical and mathematical sciences. Several forums and initiatives involving industry, government, and academia have attempted to address the issue of the ongoing difficulties of attracting students to the discipline, but with little perceptible effect. New initiatives that seek to address the issues of students and funding will be a CORE priority in 2009.

During 2008, CORE continued to work on journal and conference rankings, with much of the activity driven by requests for information from the government. A key aim is now to maintain the rankings, which are widely used overseas as well as in Australia, a challenging process that needs to balance competing special interests as well as addressing the interests of the community as a whole. A new activity in 2008 was a review of computing curriculum, which is still ongoing, with the intention that a CORE curriculum statement be used for accreditation of degrees in computer science, software engineering, and information technology. ACSW'09 includes a forum on computing curriculum to discuss this process.

CORE's existence is due to the support of the member departments in Australia and New Zealand, and I thank them for their ongoing contributions, in commitment and in financial support. Finally, I am grateful to all those who gave their time to CORE in 2008; in particular, I thank Jenny Edwards, Alan Fekete, Tom Gedeon, Leon Sterling, Vanessa Teague, and the members of the executive and of the curriculum and ranking committees.

Justin Zobel
President, CORE
January, 2009

ACSW Conferences and the Australian Computer Science Communications

The Australasian Computer Science Week of conferences has been running in some form continuously since 1978. This makes it one of the longest running conferences in computer science. The proceedings of the week have been published as the *Australian Computer Science Communications* since 1979 (with the 1978 proceedings often referred to as *Volume 0*). Thus the sequence number of the Australasian Computer Science Conference is always one greater than the volume of the Communications. Below is a list of the conferences, their locations and hosts.

2010. Volume 32. Host and Venue - Queensland University of Technology, Brisbane, QLD.

2009. Volume 31. Host and Venue - Victoria University, Wellington, New Zealand.

2008. Volume 30. Host and Venue - University of Wollongong, NSW.
2007. Volume 29. Host and Venue - University of Ballarat, VIC. First running of HDKM.
2006. Volume 28. Host and Venue - University of Tasmania, TAS.
2005. Volume 27. Host - University of Newcastle, NSW. APBC held separately from 2005.
2004. Volume 26. Host and Venue - University of Otago, Dunedin, New Zealand. First running of APCCM.
2003. Volume 25. Hosts - Flinders University, University of Adelaide and University of South Australia. Venue - Adelaide Convention Centre, Adelaide, SA. First running of APBC. Incorporation of ACE. ACSAC held separately from 2003.
2002. Volume 24. Host and Venue - Monash University, Melbourne, VIC.
2001. Volume 23. Hosts - Bond University and Griffith University (Gold Coast). Venue - Gold Coast, QLD.
2000. Volume 22. Hosts - Australian National University and University of Canberra. Venue - ANU, Canberra, ACT. First running of AUIC.
1999. Volume 21. Host and Venue - University of Auckland, New Zealand.
1998. Volume 20. Hosts - University of Western Australia, Murdoch University, Edith Cowan University and Curtin University. Venue - Perth, WA.
1997. Volume 19. Hosts - Macquarie University and University of Technology, Sydney. Venue - Sydney, NSW. ADC held with DASFAA (rather than ACSW) in 1997.
1996. Volume 18. Host - University of Melbourne and RMIT University. Venue - Melbourne, Australia. CATS joins ACSW.
1995. Volume 17. Hosts - Flinders University, University of Adelaide and University of South Australia. Venue - Glenelg, SA.
1994. Volume 16. Host and Venue - University of Canterbury, Christchurch, New Zealand. CATS run for the first time separately in Sydney.
1993. Volume 15. Hosts - Griffith University and Queensland University of Technology. Venue - Nathan, QLD.
1992. Volume 14. Host and Venue - University of Tasmania, TAS. (ADC held separately at La Trobe University).
1991. Volume 13. Host and Venue - University of New South Wales, NSW.
1990. Volume 12. Host and Venue - Monash University, Melbourne, VIC. Joined by Database and Information Systems Conference which in 1992 became ADC (which stayed with ACSW) and ACIS (which now operates independently).
1989. Volume 11. Host and Venue - University of Wollongong, NSW.
1988. Volume 10. Host and Venue - University of Queensland, QLD.
1987. Volume 9. Host and Venue - Deakin University, VIC.
1986. Volume 8. Host and Venue - Australian National University, Canberra, ACT.
1985. Volume 7. Hosts - University of Melbourne and Monash University. Venue - Melbourne, VIC.
1984. Volume 6. Host and Venue - University of Adelaide, SA.
1983. Volume 5. Host and Venue - University of Sydney, NSW.
1982. Volume 4. Host and Venue - University of Western Australia, WA.
1981. Volume 3. Host and Venue - University of Queensland, QLD.
1980. Volume 2. Host and Venue - Australian National University, Canberra, ACT.
1979. Volume 1. Host and Venue - University of Tasmania, TAS.
1978. Volume 0. Host and Venue - University of New South Wales, NSW.

Conference Acronyms

ACE. Australian/Australasian Computing Education Conference.
ACSAC. Asia-Pacific Computer Systems Architecture Conference (previously Australian Computer Architecture Conference (ACAC).
ACSC. Australian/Australasian Computer Science Conference.
ACSW. Australian/Australasian Computer Science Week.
ADC. Australian/Australasian Database Conference.
AISW. Australasian Information Security Workshop.
APBC. Asia-Pacific Bioinformatics Conference.
APCCM. Asia-Pacific Conference on Conceptual Modelling.
AUIC. Australian/Australasian User Interface Conference.
AusGrid. Australasian Workshop on Grid Computing and e-Research.
CATS. Computing - The Australian/Australasian Theory Symposium.
HDKM. Australasian Workshop on Health Data and Knowledge Management.
HIKM. Australasian Workshop on Health Informatics and Knowledge Management (former HDKM).

Note that various name changes have occurred, most notably the change of the names of conferences to reflect a wider geographical area.

ACSW and ACE 2009 Sponsors

We wish to thank the following sponsors for their contribution towards this conference. For an up-to-date overview of sponsors of ACSW 2009 and ACE 2009, please see `http://www.mcs.vuw.ac.nz/Events/ACSW2009/Sponsors`.

CityLink, New Zealand,
www.citylink.co.nz

New Zealand Computer Society,
www.nzcs.org.nz

Victoria University of Wellington,
www.victoria.ac.nz

Australian Computer Society,
www.acs.org.au

CORE - Computing Research and Education,
www.core.edu.au

Xero,
www.xero.com

Security Assessment, New Zealand,
www.security-assessment.com

Catalyst, New Zealand,
www.catalyst.net.nz

Helium, New Zealand,
www.heliumnz.co.nz

ACM Special Interest Group on Computer Science Education,
www.sigcse.org

RMIT University,
www.rmit.edu.au

Auckland University of Technology,
www.aut.ac.nz

Keynote

Contextualized Computing Education of Programming

Mark Guzdial

College of Computing
Georgia Institute of Technology
Atlanta, GA.

guzdial@cc.gatech.edu

One of the most powerful tools for improving success rates in introductory computing courses is the incorporation of context – a theme that pervades the computing lectures, assignments, and examples which relates the content to a concrete application domain.
Contextualized computing education has even allowed us to be successful with challenging audiences, such as the non-technical major. In this talk, we review why Georgia Tech has chosen to teach serious computer science to every student on campus, and then discuss research findings from several schools on the benefits and costs of contextualized computing education.

Invited Papers

A Perspective on the International Olympiad in Informatics for CS Educators

Margot Phillipps

School of Mathematics,
Lynfield College, Auckland,
New Zealand

margot.phillipps@gmail.com

Leon Sterling

Department of Computer Science & Software Engineering, University of Melbourne,
Australia

leon@cs.mu.oz.au

At the 2008 International Olympiad in Informatics held in Cairo, the Australian and New Zealand teams had their best ever performances. This talk will give details of the Informatics competition, and how teams are supported in Australia and New Zealand. Some sample informatics problems will be described. We argue that Informatics is an excellent basis for Computer Science at University and consequently it is important for CORE educators to understand and engage with the high school competition.

The BRACElet 2009.1 (Wellington) Specification

Jacqueline L. Whalley
Computing and Mathematical Sciences
Auckland University of Technology
Auckland 1020, New Zealand
jacqueline.whalley@aut.ac.nz

Raymond Lister
Faculty of Engineering and Information Technology
University of Technology, Sydney
NSW 2007, Australia
raymond@it.uts.edu.au

Abstract

BRACElet is a multi-institutional computer education research study of novice programmers. The project is open to new members. The purpose of this paper is to: (1) provide potential new members with an overview of BRACElet, and (2) specify the common core for the next data collection cycle. In this paper, BRACElet is taking the unusual step of making its study design public before data is collected. We invite anyone to run their own study using our study design, and publish their findings, irrespective of whether they formally join BRACElet. We look forward to reading their paper.

Keywords: BRACElet, novice programming, multi-institutional collaboration.

1 Introduction

In any academic discipline, it takes years to become an expert. Research across several disciplines indicates that experts, in addition to knowing more, also organize their knowledge into more sophisticated and flexible forms than novices. This enables the expert to bring the most appropriate form of knowledge to bear on solving a specific problem. For example, when asked to memorize chess board positions, novices tend to remember the position of each piece in isolation, whereas experts recognise and remember the attacking and defensive combinations of the pieces (Chase and Simon 1973).

Results from many studies of novice and expert programmers (e.g. Adelson 1984) are consistent with the findings from other disciplines – novice programmers form concrete representations based on how the code functions whereas experts form abstract representations based upon the purpose of the code.

The Leeds Group (Lister et al. 2004) gave students short pieces of code and asked them to determine what the values in the variables would be after the code had been executed. The Leeds Group found that almost all the students – even the 'top' students who answered most questions correctly – solved the problems by hand executing ('tracing') the code. In contrast, a follow up study found that academics tended to use a more sophisticated strategy to solve similar problems (Lister et al., 2006). Instead of tracing the code, the academics deduced the computation being performed by the code and then inferred the output directly from the input.

When teaching programming, lecturers frequently use diagrams to illustrate how a piece of code works. Expert programmers also frequently use diagrams to develop their understanding of an unfamiliar or buggy piece of code. In contrast, Thomas, Ratcliffe, and Thomasson (2004) found that many of their students were reluctant to use diagrams as an aid in tracing code, under circumstances that encouraged the students to use diagrams, even after students had been explicitly instructed in how to use the diagrams. Thomas, Ratcliffe, and Thomasson were led to conjecture that providing students with a specific diagrammatic abstraction of the code was not helpful because the self-development of such abstractions is intrinsic to developing an understanding of code.

If students cannot trace through code, how can those students write code? Traynor, Bergin and Gibson (2006) interviewed students who had completed exercises in both code tracing and code writing. One of the students who did relatively better on the writing tasks than the tracing tasks explained that the way in which writing tasks are graded helps students to gain a passing mark with only a weak grasp of what the code needs to do:

> "*… you usually get the marks for making the answer look correct. Like if it's a searching problem, you put down a loop, and you have an array and an if statement. That usually gets you the marks … Not all of them, but definitely a pass*".
>
> Student quoted in Traynor, Bergin and Gibson (2006)

The implication of the above student quote is that, when an academic grades code written by a student, there is a danger that the teacher will subscribe to the student a depth of understanding that the student does not have.

In our own teaching, we have noticed behaviours indicating that students do not understand their own code, such as:

- Attempting to debug code, sometimes for long periods of time, by 'random mutation'.
- Introducing new bugs as they attempt a superficial and incorrect fix to an existing bug. Ginat (2008) noted a similar behaviour.
- Asking the teacher for help with a bug, after the student has worked hard to find the bug, and the teacher identifies the bug with a single reading of the code.

Copyright © 2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

• Being unable to explain their own code to the teacher. Thomas, Ratcliffe, and Thomasson also relate such an anecdote.

Our own teaching, and the literature on novice programmers, has led the authors (and our collaborators in the BRACElet project) to the belief that there is hierarchy of skills associated with programming. At the bottom of the hierarchy is knowledge of basic programming constructs (e.g. what an "if" statement does). At the top of the hierarchy is the ability to write non-trivial, correct code using those programming constructs. The intermediate levels of the hierarchy are manifested in abilities such as:

- The ability to trace code, especially when there are too many variables to maintain in short term memory.
- The ability to understand/use diagrams and other abstractions of code.

More formally, our research goals are encapsulated in the following questions:

- Are there intermediate skills and knowledge in programming? If so, then …
- Is it possible to assess explicitly a student's grasp of these intermediate skills and knowledge?
- Is it possible to teach explicitly these intermediate skills and knowledge?

These are the research questions that have driven the BRACElet project. In the next section, we review past work on BRACElet. Subsequent sections of the paper then define the next iteration of work in the project. On the basis of this description of BRACElet, we invite others to join the project, or to at least conduct their own related studies.

2 The Brief History of BRACElet

The BRACElet project is a multi-institutional study of novice programmers. The project commenced in 2004 and to date 10 workshops have been held in Australasia. A useful summary of the first eight workshops has been provided by Clear et al. (2008c) and the reader is referred to that paper for a more detailed history of the project.

While BRACElet is a research project, the intention is that the project should remain close to education practice. Thus, most BRACElet data is collected via end–of–semester exams that students take at the participating educational institutions.

The following subsections summarise the evolution of the BRACElet project, through the workshops and in the papers that have emerged from the project. These subsections only offer a brief summary, and readers are referred to the earlier BRACElet papers for a comprehensive account.

2.1 The First BRACElet Paper

The first BRACElet workshop, in December 2004, began with a review of the results from the then recently published Leeds Group (Lister et al. 2004). The workshop participants felt that the Leeds Group study was not sufficiently based upon learning theories or educational models.

The workshop participants selected the revised version of Bloom's taxonomy (Anderson et al., 2001) as an educational model for test question development, and then devised several questions that mapped to different parts of that taxonomy. None of the questions required the students to write code, but instead tested their ability to reason about code and also reason about associated abstractions, such as flow charts. These questions were then included in the end–of–first–semester exams that students attempted at some of the participating New Zealand institutions, in June 2005. Analysis of the data from those exams began at the second BRACElet workshop, in July 2005. The first BRACElet paper (Whalley et al. 2006) was a consequence of the analysis started at that second workshop.

One of the findings reported in that paper concerned the revised Bloom's taxonomy. The results are summarised in Figure 1. Students tended to do best on questions from the *Understand* level of the taxonomy, which was the lowest level of taxonomy for which questions were designed, and which required the least abstract reasoning. Students tended to do less well on the more abstract *Apply* questions, and students did least well on *Analyse* questions, which was the highest level of the taxonomy for which BRACElet participants designed questions, and which required the most abstract reasoning (but did not require students to write code).

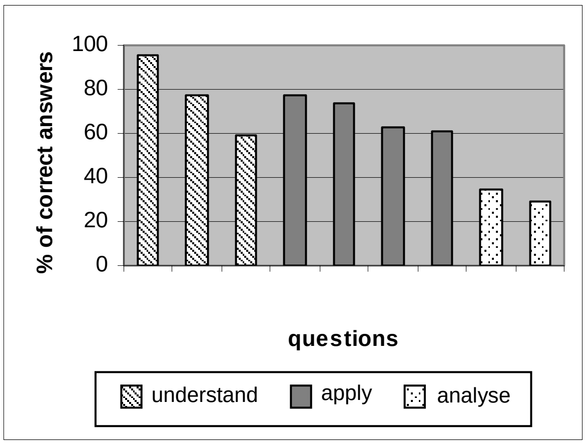


Figure 1: Novice performance on questions classified by Bloom's revised category (Whalley et al., 2006).

The second workshop added another educational model to the project, the SOLO taxonomy (Biggs and Collis 1982). This taxonomy was introduced to analyse data for the question shown in Figure 2. The student answers to that question were placed into one of five categories, based on the SOLO taxonomy, as shown in Table 1.

In plain English, explain what the following segment of Java code does:

```
bool bValid = true;

for (int i = 0; i < iMAX-1; i++){
  if (iNumbers[i] > iNumbers[i+1])
    bValid = false;
}
```

Figure 2. An 'explain in plain English' question

Using the scores that the students achieved on all the BRACElet questions in the exam, except the 'explain in plain English' question, the students were broken into four quartiles. Figure 3 shows the types of SOLO answers that students in each of those quartiles gave to the 'explain in plain English' question. Students in the top two quartiles were far more likely to give a relational answer (i.e. an answer giving a summary of the purpose of the code) than students in the lower two quartiles.

SOLO category	Description of student's answer
Relational	A summary of the purpose of the code. For example, "*checks if the array is sorted*".
Multistructural	A line by line description of all the code. Some summarisation of individual statements may be included.
Unistructural	A description of one portion of the code (e.g. describes the *if* statement).
Prestructural	Shows substantial lack of knowledge of programming constructs or is unrelated to the question.
Blank	Question not answered.

Table 1: The SOLO Categories for the students' answers to the 'explain in plain English' question

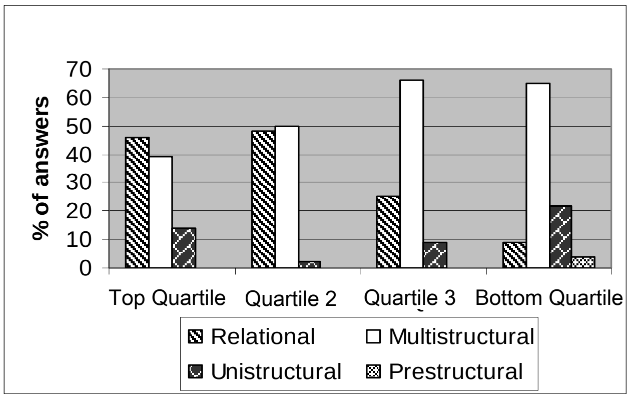


Figure 3. Performance of students on the 'explain in plain English' question (Whalley et al., 2006).

In the conclusion of that first BRACElet paper, the following observations were made:

It appears likely that programming educators may be systemically underestimating the cognitive difficulty in … [exam questions] *… for assessing programming skills of novice programmers. … The stages through which novice programmers develop … and the time that this development process may take… may have been significantly underestimated ... Students who cannot read a short piece of code and …* [summarise the computation performed by that code]*… are not well equipped intellectually to write code of their own.*

2.1.1 Reflection: The Two-Task Approach

BRACElet has always been focussed on identifying aspects of novice programmers that transcend institutional boundaries. The most obvious potential difference between cohorts of novice programmers at different institutions is the ability of the 'typical' student at each institution. Suppose we are studying students at two different Institutions, I1 and I2, where I1 is a more prestigious educational institution than I2. If we give the same programming question, Q1, to students at both institutions, it is likely that, on average, students from I1 will do better on Q1 than students from I2. The Leeds Group faced the same problem and made the following observation:

There are trends in the data across institutions. In general, a question that is substantially more difficult for students at one institution is also more difficult for students at other institutions.

Lister et al. (2004, p. 126).

The BRACElet project has taken that observation and used it as a method of analysis, referred to as the two-task approach. Suppose students at I1 and I2 are given two questions Q1 and Q2. While students at I1 might do better on both Q1 and Q2 than students at I2, BRACElet participants look for patterns in the relative performance of students on the two questions. For example one pattern may be that students at both I1 and I2 do better, on average, at Q1 than Q2.

The two-task approach generalizes to N tasks. The first BRACElet paper (Whalley et al. 2006) used a three task approach when analyzing questions from different levels of Bloom's taxonomy. Even though students at I1 might outperform students from I2 at each level of the revised Bloom's taxonomy, students within each of those institutions will tend, on average, to perform better on *Understand* questions than *Apply questions*, and better on *Apply* questions than *Analyse* questions.

The two-task approach can also be used another way. One task (or one set of tasks) is used to place students from different institutions into two or more categories. The students in a single category are then considered to be equivalent, even though they come from different institutions. All students are also given a second task (or a second set of tasks). The statistical performance on the second task by students in one category is compared to the statistical performance of students in another category. This two task approach was used to perform the SOLO analysis of the above 'explain in plain English' question. Nine multiple choice questions were used as the task set 1. Students were placed into four quartiles (i.e. categories), based upon their score on those nine multiple choice questions. The analysis then focussed on differences between the quartiles in the frequencies of the various SOLO levels.

The two-task approach reduces the need to collect data describing the educational and institutional backgrounds of students. For example, if two students are both placed into the same category based on their performance on task 1, BRACElet participants are not concerned if one of those students had programmed prior to arriving at university, and the other had not; nor are BRACElet participants concerned if one of those students is from an elite university and the other is not.

2.2 Follow up to the First Paper

A number of subsequent BRACElet papers elaborated upon the work of the first BRACElet paper. One of those papers (Whalley 2006) looked for any effect due to programming language. BRACElet participants did not teach the same programming language, so the questions were translated into the appropriate language for each institution. Whalley's analysis revealed no significant difference between the mean total score of students at one institution who completed the questions in the Delphi programming language and the students at another institution who completed the test in C++.

Another follow up study (Lister et al. 2006) asked academics to 'think out loud' as they solved problems from the Leeds Group study that were similar to problems from the first BRACElet paper. That follow up study found that academics actively seek to abstract beyond the concrete code, whereas the Leeds Group had found that novices did not seek to abstract.

2.3 Code Classification Questions

Thompson et al. (2006) explored a different kind of assessment question that asked students to make explicit statements about their understanding of code. In an exam question, they presented students with four segments of code. All four pieces of code contained a loop with an "if" statement inside the loop. Two of the segments used a 'for' loop and other two segments used a 'while' loop. The students were not told that two of the pieces of code found the minimum value in an array, and the other two pieces found the maximum. Students were asked to iteratively place the four pieces of code into two groups, so that code pieces inside a group were similar in some way and different from the code pieces in the other group. Thompson et al. found that many students grouped the code pieces based on syntactic features (e.g.; "uses a for loop vs. uses a while loop", but failed to identify the more abstract functional similarities that indicate an ability to abstract the purpose of code (e.g. "finds the minimum vs. finds the maximum").

Thompson et al. also compared student responses to the 'explain in plain English' questions with their performance on the new classification question. Thompson et al. found that if a student gave a relational response to an explain in plain English question they were more likely to provide responses that identified functional similarities when answering a classification question. It appears that both questions types are reasonably reliable indicators of whether a student can extract meaning from a short piece of code.

The analysis presented in this paper highlighted a number of interesting questions for future study, including:

- Can a novice programmer's ability to understand and write programs be fostered through the use of variations in code and through code classification questions?
- Would a student who is able to identify the purpose of a code segment (a relational classification) in one language also recognise a code segment in a different language that has the same logic structure and achieves the same task?

2.4 Parson's Problems

A common criticism of having students solve pen-and-paper code writing problems in an exam is that it is a non authentic task and it is unreasonable to expect students to program without the support of a compiler.

Parson's puzzles are an alternative to code writing. In such puzzles, students are presented with jumbled lines of code which they are required to place into the correct order so that the code performs a prescribed task (Parsons and Haden 2006). The same type of puzzle was also developed independently for the "Head First' textbooks (Sierra and Bates 2005).

In more recent iterations of the BRACElet study (Whalley and Robbins 2007b, Lopez et al. 2008) a simple Parson's puzzle was used in an exam. The students performed much better on Parson's puzzles than code writing questions despite the fact that the students had never seen such a problem before. The authors suggested that Parson's puzzles might be a bridge between full code writing tasks and code reading tasks.

Denny, Luxton-Reilly and Simon (2008) furthered the study of student performance on Parson's puzzles and enlisted a student feedback loop in order to refine the way in which Parsons puzzles are presented in an examination. A notable correlation between Parson's scores and code writing scores was found. They recommend the use of Parson's puzzles in assessment:

> *"Parsons problems are easier and more reliable to mark than code writing, provide an opportunity to test student misconceptions more specifically than code writing, yet they appear to require the same set of skills (as analyzed by correlation in marks achieved). This makes them an excellent alternative to traditional code writing questions"*

2.5 The Traffic Light Conjecture

Philpott, Robbins and Whalley (2007) analysed student responses to code tracing and 'explain in plain English' questions. A link was found between well developed tracing skills and the ability to think relationally about code. Conversely, students who were unable to arrive at a correct answer when tracing code were unable to reason about code segments, and instead focused on a single line of code within code segments. These findings led to the 'traffic light' conjecture – that a student's degree of mastery of code tracing tasks indicates their readiness or ability to be able to reason about code. It was proposed that a student has reached a 'green light' for relational thinking if they have mastered the ability to trace code. If they are able to trace reliably most of the time they have reached an 'orange light'. If students trace code correctly less than 50% of the time they are still at a 'red light' and they are only able to explain code in a unistructural way.

This study highlights the use of the two-task approach to study a hierarchy of programming skills. Here, tracing is task 1, and only students who demonstrate a particular level of competence in task 1 are capable of performing task 2.

2.6 The Relationship to Code Writing

From the outset of the BRACElet project, we intended to investigate the relationship between code reading and code writing. As educators, it was our intuition that code tracing is easier than code writing and that the ability to read code is a precursor to code writing, but there is little direct evidence to support that intuition.

Sheard et al. (2008) gave students three ‘explain in plain English’ questions in an end-of-first-semester exam, and also required the students to write code. The level of SOLO response to the ‘explain in plain English’ questions correlated positively with performance on writing tasks. However, correlation is not evidence of causality, and there remains much work to do before it can be said with confidence that the capacity to answer ‘explain in plain English’ questions involves skills that are a precursor to code writing.

2.7 The Path to Abstraction

Lopez et al. (2008) analysed student responses to an end-of-first-semester exam, looking for evidence of a hierarchy of programming skills. They looked for statistical relationships between the questions in the exam; specifically how well non-writing questions predicted student performance on a code writing question. A stepwise regression, with performance on code writing as the dependent variable, was used to construct a path diagram. The diagram suggests the possibility of a hierarchy of programming related tasks. At the bottom of their regression hierarchy were exam questions that required little more than knowledge of programming constructs. At intermediate levels of the regression hierarchy were “explain in English” questions, Parson’s puzzles, and the tracing of iterative code.

2.8 Doodles

McCartney et al. (2007) analysed student annotations on a test sheet (‘doodles’) and investigated the relationship between those doodles, the difficulty of the questions and student performance. They found that students who doodled performed better on the Leeds problem set.

Whalley, Prasad and Kumar (2007a) also looked at student annotations using a new problem set. They used the SOLO taxonomy to investigate the level of reasoning that students achieved when answering a short answer question and related this to the type and number of annotations they made. Whalley Prasad and Kumar found that despite encouraging students to doodle and teaching tracing, almost two thirds of the students turned in an exam paper on which there were no annotations. While higher achievers were more likely to doodle there seemed to be no relation between the use of annotations and the SOLO responses to ‘explain in plain English’ questions.

Whalley, Prasad and Kumar also found that certain types of questions are more likely to elicit certain doodle types. They argued that certain doodles actually arise because of the constructs in the code rather than the question type. For example, it could be argued that tracing would be a more appropriate strategy when the code fragment contains a loop.

This study raised several questions which have still not been investigated:

- Why in the majority of circumstances do students not annotate their scripts?
- When students do doodle what motivates them to do so?
- When students do annotate are they using doodles that are appropriate for the question type?

2.9 Reasoning with Diagrams

Lister (2007) used a two-test approach to study student performance on an end-of-first-semester exam. Task 1 comprised four multiple choice questions, in which students were asked to determine the value in a particular variable, after a given piece of code had finished executing.

In the second task, students were given eight multiple choice questions, but questions of a different nature from those in task 1. For each task 2 multiple choice question, students were given a set of diagrams that described an algorithm, and they were also given code that implemented that algorithm, with one or two lines missing. In each of the eight questions, students had to choose the appropriate line(s) from four options provided. Of the students who scored a perfect 4 on Task 1, approximately 20% scored 5 or less out of a possible 8 on Task 2. Those 20% were accomplished concrete reasoners about code – having scored a perfect 4 on task 1 – but those students did not manifest the more abstract skill of relating code and diagrams. This study is evidence that the ability to reason about algorithms diagrammatically is a skill that is higher on the hierarchy than the ability to trace code.

This study is consistent with the work of Thomas, Ratcliffe, and Thomasson (2004) that was discussed in the introduction of this paper. They found that many of their students were reluctant to use diagrams as an aid in tracing code – which is not surprising for any student who cannot reason about algorithms diagrammatically.

2.10 Developing the Research Methods

The BRACElet project has made extensive use of two taxonomies, the revised Bloom’s taxonomy and the SOLO taxonomy. However, in doing so, BRACElet participants have had to come to a mutual understanding of how these taxonomies apply to the work of novice programmers. This has not been easy – indeed BRACElet participants are yet to develop a strong consensus on how these taxonomies are best applied to novice programmers. Thompson, et al. (2008) represents our best attempt thus far to apply a consistent interpretation with concrete exemplars of the revised Bloom’s taxonomy, and Clear, et al. (2008b) does the same for the SOLO taxonomy.

3 The BRACElet Guiding Principles

A multi-institutional collaboration like BRACElet can only be successful if the participants have bought into the principles on which the project is founded. In this section we supply a brief summary of the guiding principles for the BRACElet paper. Many of our guiding principles have their foundation in earlier multi-institutional projects (Fincher et al. 2005).

A detailed account of the practices and rules for BRACElet collaborators are in Whalley and Robbins (2007b), Whalley, Clear and Lister (2007c) and Clear et al. (2008a). Anyone who is contemplating joining BRACElet should read those documents.

3.1 Belief in a strong teaching–research nexus

BRACElet members believe in a strong teaching–research nexus. We see our work as an evidence-based and research-informed way of improving our teaching. We see BRACElet as:

"… the development of theory or understanding as a by-product of the improvement of real situations, rather than application as a by-product of advances in 'pure' theory."

(Carr and Kemmis, 1986, p. 28).

As part of this commitment to research, BRACElet members debate the evidence, and not their folk-pedagogic intuitions.

3.2 Assessing the student always comes first

One of the core guiding principles is that assessing the students always comes first and the research must be conducted without compromising the course.

3.3 The Common Core

Placing the student first means that it is impossible to run exactly the same experiment (i.e. use exactly the same exam questions) in every participating institution. Instead, before each data gathering phase, BRACElet members agree upon a 'common core', which is a minimal set of properties that must be present in all their exams. For example, past common core definitions have specified conditions like 'all exams will contain at least one explain in plain English question'. Such a specification leaves each institution with many degrees of freedom, such as what programming constructs will be in their choice of code, how long the code will be, and whether or not to give the students practice in this type of question prior to the exam. BRACElet participants were even free to use a different phrase than 'explain in plain English'. For example, in one exam (Sheard et al. 2008) the students were asked to supply a meaningful name for a method.

The specification of the common core for the next phase of BRACElet data gathering – the 2009.1 (Wellington) specification – is given later in this paper.

3.4 Repetition for Robustness

Rather than weakening BRACElet, the common core makes our findings more robust. When non-identical tests are used at different institutions, but we see similar patterns in the results, we have a research outcome that has implications beyond the classrooms of the BRACElet collaborators. Fincher and Petre (2004) explained the importance of this type of repetition:

"… repetition can show how consistent the outcomes of a given study are across different related tasks, across different environments, across different related contexts."

3.5 The Rules of Engagement

What really drives BRACElet is a set of "Rules of Engagement" that cover the brainstorming or ideas phase and the post-brainstorming phase (Whalley, Clear and Lister, R. 2007c). The basic principle of these rules is that ideas are the easy part. BRACElet recognises that the hard graft is done gaining ethics clearance, drafting the actual final instruments, collecting the data, doing the analysis and writing the paper. When we meet we share ideas freely. All meeting participants are free to use those ideas, without any obligation, such as co-authorship, to the person who first uttered the idea.

3.6 The Publication Protocol

BRACElet members are not automatically authors of any BRACElet paper. In fact, most BRACElet papers are written by subgroups. Our policy has been that all co-authors must have made a substantial contribution to at least two of the following four activities:

- Conception or design
- Data collection and processing
- Analysis and interpretation of data
- Writing substantial sections of the paper (e.g. synthesising findings in a literature review or a findings/results section)

Also, everyone who is listed as an author should be able to defend the paper as a whole (although not necessarily all the technical details).

With the publication of this paper, there is a relaxation of one the above authoring requirements, which will apply as an experiment in the next data gathering phase, after which it will be considered as a permanent change to the rules. The relaxation is that anyone who contributes data that is collected in an exam from their students, but does not contribute on any of the other points, will be a co-author on at least one BRACElet paper that uses that data, probably the first paper. Note that contributing data implies meeting the institutional ethical clearance processes, adapting the questions to the specific exam, and providing the data in a form that is useful to other BRACElet participants.

Anyone who attended both days of the BRACElet workshop in Sydney, in September 2008, is deemed to have contributed to conception and design for the next iteration of the BRACElet research cycle.

3.7 Membership / Recruitment

BRACElet welcomes new members. Indeed, the need to repeatedly explain and justify our research to new members actually tests and strengthens our research. At every BRACElet meeting, there is little distinction between reviewing the project and bringing the new people up to speed – if we cannot explain our research to new people, what use is that research to anyone?

Joining BRACElet obliges the new member to abide by our guiding principles, especially the rules of engagement and the publication protocol. Also, members need to attend some of the project meetings, but by no means all meetings. Almost all members begin their membership via attending their first meeting.

New BRACElet members *must* complete the appropriate human ethics clearance processes at their institution (known as 'IRB' to most Americans). For anyone who would like to join BRACElet, but who has no prior experience with getting such approval, we can help.

3.8 The Open Research Plan

Prior to the publication of this paper, BRACElet has traditionally kept each research plan private among its members until the data has been collected, analysed and a paper has been written. With this paper, BRACElet begins an experiment with a different approach, where a specification of the research plan is published prior to data collection, analysis and publication.

With the publication of this paper, our research plan is in the public domain. Anyone is free to collect their own data, analyse and publish it. In fact, we encourage people to do so. Neither the authors of this paper, nor existing members of BRACElet, have any claim to co-authorship with anyone who does so, nor do we have any claims to the data of those people. However, we do ask that people observe certain professional courtesies. They should:

(1) Cite and summarise this paper in their paper(s).

(2) Not claim to be part of the BRACElet project, unless they have our explicit approval to do so. Instead, they should refer to their work as being based upon the BRACElet 2009.1 (Wellington) specification. We reserve all naming rights to "BRACElet".

(3) Be explicit in their paper(s) about any alteration they made to the 2009.1 (Wellington) specification. They should NOT use a new version number to refer to their changes. We reserve naming rights to version numbering.

4 The 2009.1 (Wellington) Common Core

This section specifies the BRACElet 2009.1 (Wellington) common core, which comprises three parts:

(1) Basic Knowledge & Skills

(2) Reading / Understanding

(3) Writing

These three parts are described in detail the next three subsections.

If students complete the different parts of the common core at different times, the elapsed time between doing the first part and doing the last part should be no more than 1 week. Also, record the relative times when each part was done (e.g. 'part 2 was done 3 days after part 1').

4.1 Basic Knowledge & Skills

The purpose of this part of the common core is two-fold. First, it establishes that students understand the programming constructs (e.g. how an "if" statement or a "while" loop works). Second, it establishes that students have mastered relatively concrete skills, such as tracking variable updates as the student traces through code. Ideally, BRACElet participants should use several questions to test students on their basic knowledge and skills.

Thus far in BRACElet, some form of tracing question has always been used to establish that students understand the programming constructs. As a guiding (but non-mandatory) principal for tracing questions, there should be three types of questions:

(1) Non-iterative (and non-recursive) tracing questions, where several programming constructs may be tested in the one question;

(2a) Iterative tracing questions, without control logic within the loop, and with a very small number of variables, perhaps no more than a loop control variable plus one other variable. For example, a loop that sums the elements of an array meets those criteria (but it might be wise to avoid using a variable called "sum", otherwise students might guess the answer). This type of question establishes whether students understand the iterative construct used.

(2b) Iterative tracing questions that have control logic within the loop (e.g. the code in Figure 2) which establishes whether students can track variable updates as they trace that code. The code in at least one of these tracing questions should be of similar complexity to the code in one question from part 2 of the common core. Also, the code in at least one of these tracing questions should be of similar complexity to the code in one question from part 3 of the common core

Most BRACElet participants use multiple choice questions for tracing questions, but free response is also allowed.

These questions should not contain buggy code. That is, the code should not contain simple bugs, such as a loop that terminates either one iteration too early or too late. As a guiding principal, if an expert programmer was to attempt these questions hastily, they should not be tricked by a simple bug. Note, however, that nonsense code is permissible (i.e. code that has no purpose that an expert programmer could easily identify and describe).

BRACElet members are left free to decide whether the code will use meaningful variable names. Members who use meaningful names for parts 2a and 2b are warned not to use a piece of code for which a student might then guess the function of the code, and thus answer the question without tracing the code.

New BRACElet participants are encouraged to reuse questions published in past BRACElet papers – it is unlikely that any of your students are reading the BRACElet papers.

The common core does not specify on which programming constructs the knowledge of students should be tested – that is determined by the syllabus at each participating institution. The common core merely specifies that this part of the common core should test all the programming constructs used in the other two parts of the core.

4.1.1 Anticipated Analysis

The primary purpose of this first part of the common core is to act as task 1 in two-task analysis. However, we would also like to build upon the earlier BRACElet work on the doodles students make when tracing code (Whalley, Prasad and Kumar 2007a). Therefore, BRACElet participants should retain the exam scripts / answer booklets of their students, and be prepared to show those materials to other BRACElet members. This may be an issue for human ethics clearance. Students should be explicitly encouraged on the exam paper to show their working in a designated place (e.g. on the same page where a question occurs in the exam paper; perhaps in a box on that page if there is more than one question on the page).

When collecting data an established anonymous script scanning and data source identification protocol should be adhered to. All participants must follow this protocol, usually enforced by an institutions human ethics approval, which is created to safeguard the privacy of students and institutions.

4.2 Reading / Understanding

The purpose of this part of the common core is to establish whether students deduce the purpose of a piece of code from reading the code. BRACElet members should use at least one question based on any of the following three question types:

(a) Explain in plain English. BRACElet members are free to use an instruction to students other than ‘explain in plain English’.

(b) Parson’s problems.

(c) Code Classification Questions, as in Thompson et al. (2006). However, if a BRACElet participant elects to use this type of question, they should use at least one of the other question types (as this type of question is the least evaluated and therefore the most risky of the three types).

Any code used for an explain in plain English question should not have been used in a part 1 tracing question. Otherwise, there is a danger that students might guess a SOLO relational response from the output of the code, when our aim is to see if they can produce a SOLO relational response by reading the code. It follows that at least some of the variables in any code should be left non-initialized; otherwise the student might trace the code. One way of doing this is to present the code as a method, with some of the variables declared as part of the method header.

The code used in at least one of these reading/understanding questions should be of similar complexity to at least one part 1 tracing task. Also, the code used in at least one of these reading/understanding questions should be of similar complexity to at least one part 3 writing task.

For repetition and robustness, new BRACElet participants are again encouraged to reuse questions published in past BRACElet papers.

4.2.1 Anticipated Analysis

BRACElet participants are encouraged to include several reading/understanding questions in their exam (of the same type, or of different types). With multiple questions we can examine the following issues:

(a) For ‘explain in plain English’ questions, do students tend to provide responses at the same SOLO level for code of similar complexity? The work by Sheard et al. (2008) suggested that this was the case.

(b) For two ‘explain in plain English’ questions of differing levels of complexity, do students tend to provide a response at a lower SOLO level for the more complex code? Again, the work by Sheard et al. (2008) suggested that this was the case.

(c) For a given level of code complexity, is there a statistical relationship between the SOLO level of a student response to an ‘explain in plain English’ question, and a code classification question? The work by Thompson et al. (2006) suggested that this was the case.

(d) For a given level of code complexity, can students who provide multistructural responses to ‘explain in plain English’ questions solve Parson’s Problems?

4.3 Common Core 3: Writing

Most programming exams already require students to write code, and BRACElet has only one additional requirement for such writing questions – at least one code writing task should be of similar complexity to one tracing task (i.e. from part 1 of the core), and at least one code writing task should be of similar complexity to one reading/understanding task (i.e. part 2 of the core).

4.3.1 Anticipated Analysis

BRACElet regards code writing ability as the dependent variable. That is, if programming is indeed a hierarchy of knowledge and skills, then code writing is at the top of the hierarchy, and our analysis will continue our earlier work (e.g. Lopez et al. 2008) of searching for statistical relationships between code writing and what we believe are precursor skills for code writing.

4.4 Other Data to be Collected

Where it is possible, collect the gender of every student from whom data is collected.

Where data is collected from volunteers, record the age of the volunteer. If data is collected via an exam given to the class, it is only necessary to be able to describe the age range of most students (e.g. “almost all students are between 18 and 21 years of age”).

Where data is collected from volunteers, record the stage they have reached in their formal study of programming at your institution (e.g. ‘eight weeks into their second semester of learning programming’). Also record whether the degree is undergraduate or postgraduate.

4.5 2009.1 (Wellington) Membership

To become a member of BRACElet, data may be collected either through an exam given to students, or as a paper-based test given to student volunteers, or through a ‘think out loud’ protocol with either students or colleagues. The data collected is to be made available to all members of BRACElet.

In cases where data is gathered as part of assessment, or by volunteers completing a written test, data from at least 20 students is required, to enable statistical analysis.

Where it is not possible to collect data from 20 students, BRACElet membership can be obtained by conducting 'think out loud' sessions, with student volunteers or with colleagues.

5 Conclusion

Researchers who work within paradigms have agreed approaches and standards of evidence that allow them to work for long periods within their respective institutions. Peer reviewed papers are their primary means of communicating with researchers in other institutions. Computing education research has not yet developed its own paradigms. We do not have strong community agreement on research approaches and what constitutes evidence. Consequently, BRACElet participants have deliberately chosen to work more closely than is usual in research, across institutional boundaries, because BRACElet is as much about devising research approaches, and debating what constitutes valid evidence, as it is about studying novice programmers. We invite you to join the project. Even if you choose not to join BRACElet itself, you are welcome to execute the research plan we have published in this paper – our plan is now in the public domain, so anyone is free to use it. We look forward to reading about your results.

Acknowledgements

The authors thank their collaborators on the BRACElet project, especially our co-leader Tony Clear. Some of the past work on BRACElet has been funded by ACM SIGCSE Special Projects Grants awarded to Jacqueline Whalley and Tony Clear. Some of the current work on BRACElet is funded by an Associate Fellowship awarded to Raymond Lister and Jenny Edwards from the Australian Learning and Teaching Council (formerly the Carrick Institute).

References

Adelson, B. (1984): When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, **10**(3): 483-495.

Anderson, L.W., Krathwohl, D.R, Airasian, P.W., Cruikshank, K.A., Mayer, R.E., Pintrich, P.R., Raths, R. and Wittrock, M.C. (Eds) (2001): *A Taxonomy for Learning, Teaching and Assessing. A Revision of Bloom's Taxonomy of Educational Objectives*. New York, Addison Wesley Longman, Inc.

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.

Carr, W., and Kemmis, S. (1986): *Becoming critical: education knowledge and action research*. Lewes: Falmer Press.

Chase, W. C., and Simon, H. A. (1973): Perception in chess. *Cognitive Psychology*, **4**: 55-81.

Clear, T., Edwards, J., Lister, R., Simon, B., Thompson, E. and Whalley, J. (2008a). The teaching of novice computer programmers: bringing the scholarly-research approach to Australia. *Proc. Tenth Australasian Computing Education Conference (ACE 2008)*, Wollongong, NSW, Australia. Simon and Hamilton, M., Eds., CRPIT, **78**:63-68, Australian Computer Society, Darlinghurst, Australia.

Clear, T., Whalley, J., Lister, R., Carbone, A., Hu, M., Sheard, J., Simon, B., Thompson, E. (2008b): Reliably Classifying Novice Programmer Exam Responses using the SOLO Taxonomy. *Proc. 21st Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2008)*, Auckland, New Zealand. Samuel Mann and Mike Lopez., Eds., 23-30.

Clear, T., Philpott, A., Robbins, P., and Simon. (2008c): Report on the Eighth BRACElet Workshop (BRACElet Technical Report No. 01/08). Auckland: Auckland University of Technology.

Denny, P., Luxton-Reilly, A. and Simon, B. (2008): Evaluating a New Exam Question: Parsons Problems. *Proc. of the 2008 International Workshop on Computing Education Research (ICER '08)*, Sydney, Australia. ACM Press, New York, NY.

Fincher, S, and Petre, M. (2004): *Computer Science Education Research*, Taylor & Francis.

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. *Proc. International Computing Education Research Workshop*. Seattle, USA, 111-121.

Ginat, D. (2008): Learning from wrong and creative algorithm design. *Proc. of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*, Portland, OR, USA, 26-30, ACM Press, New York, NY.

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.

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. *Proc. of the 11th Annual SIGCSE Conference on innovation and Technology in Computer Science Education (ITICSE '06)*, Bologna, Italy, 118-122 ACM Press, NY.

Lister, R. (2007): The Neglected Middle Novice Programmer: Reading and Writing without Abstracting. *Proc. of the 20th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ'07)*, Port Nelson, New Zealand, Mann, S. and Bridgeman, N., Eds, 133-140.

Lopez, M., Whalley, J., Robbins P. and Lister, R. (2008): Relationships between reading, tracing and writing skills in introductory programming. *Proc. of the 2008 International Workshop on Computing Education Research (ICER '08)*, Sydney, Australia. ACM Press, New York, NY.

McCartney, R., Moström, J. E., Sanders, K. and Seppala O. (2004): Questions, Annotations, and Institutions: observations from a study of novice programmers. *Proc. Of the Fourth Finnish/Baltic Sea Conference on Computer Science Education*, Koli, Finland, 11-19.

Parsons, D. and Haden, P. (2006): Parsons' programming puzzles: a fun and effective learning tool for first programming courses. *Proc of the 8th Australian Conference on Computing Education*, Hobart, Australia, D. Tolhurst and S. Mann, Eds., CRPIT, **78**: 157-163.

Philpott, A, Robbins, P., and Whalley, J. (2007): Accessing The Steps on the Road to Relational Thinking. *Proc. 20th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2007),* Mann, S and Bridgeman, N., Eds, Nelson, NZ, 286.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., and Whalley, J. (2008): Going SOLO to Assess Novice Programmers. *Proc. of the 13th Annual SIGCSE Conference on innovation and Technology in Computer Science Education (ITICSE '08),* Madrid, Spain, ACM Press, New York, NY, 209-213.

Sierra, K. and Bates, B. (2005): *Head First Java,* 2nd Edition, O'Reilly Media, Inc.; 2nd edition.

Thomas, L., Ratcliffe, M., and Thomasson, B. (2004): Scaffolding with object diagrams in first year programming classes: some unexpected results. *SIGCSE Bulletin*, **36**(1):250-254.

Thompson, E., Whalley, J., Lister, R. and Simon, B. (2006): Code Classification as a Learning and Assessment Exercise for Novice Programmers. *Proc. of the 19th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2006)*, Wellington, New Zealand, 291-298.

Thompson, E., Luxton-Reilly, A., Whalley, J., Hu, M., and Robbins, P. (2008): Blooms Taxonomy for CS Assessment. *Proc. Tenth Australasian Computing Education Conference (ACE2008),* Wollongong, Australia, Simon & M. Hamilton, Eds., CRPIT, **78:** 155-161.

Traynor, D., Bergin, S., and Gibson, J. P. (2006): Automated assessment in CS1. *Proc. of the 8th Australian Conference on Computing Education - Volume 52,* D. Tolhurst and S. Mann, Eds., Hobart, Australia, *ACM International Conference Proceeding Series,* **165:** 223-228. Australian Computer Society, Darlinghurst, Australia,

Whalley, J., Lister, R., Thompson, E., Clear, T, Robbins, P., and Prasad, C. (2006): An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Proc. of the 8th Australian Conference on Computing Education - Volume 52,* D. Tolhurst and S. Mann, Eds., Hobart, Australia, *ACM International Conference Proceeding Series,* **165:** 243-252, Australian Computer Society, Darlinghurst, Australia.

Whalley, J. (2006): CSEd research instrument design: the localization problem. *Proc. 19th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2006),* S. Mann & N. Bridgeman, Eds., Wellington, NZ, 307-312.

Whalley, J., Prasad, C. and Kumar, P. K. A. (2007a): Decoding Doodles: Novice Programmers and Their Annotations. *Proc. Ninth Australasian Computing Education Conference (ACE 2007),* Ballarat, Vic., Australia. Mann, S. and Simon, Eds., CRPIT, **66**: 171-178.

Whalley, J. L., and Robbins, P. (2007b): Report on the Fourth BRACElet Workshop. *Bulletin of Applied Computing and IT*, **5**(1), Retrieved June 7, 2007 from http://www.naccq.co.nz/bacit/0501/2007Whalley_BRACELET_Workshop.htm

Whalley, J., Clear, T. and Lister, R. (2007c): The Many Ways of the BRACElet Project. *Bulletin of Applied Computing and IT*, **5**(1), Retrieved June 7, 2007 from http://www.naccq.co.nz/bacit/0501/2007Whalley_BRACELET_Ways.htm

PANEL

Second Life Panel

Clare Atkins

School of Information Technology
Nelson Marlborough Institute of Technology
New Zealand

clare.atkins@nmit.ac.nz.

Scott Diener

IT Services, Academic & Collaborative Technologies
The University of Auckland
New Zealand

s.diener@auckland.ac.nz

Nauman Saeed

Faculty of Information & Communication Technologies
Swinburne University of Technology
Australia

nsaeed@ict.swin.edu.au

There are many interesting uses of the virtual world in Second Life for Computing Education. The panel members are involved in tertiary education and use Second Life as their medium, both for teaching and research purposes. However, the session is an open forum, where they will demonstrate their work in Second Life, and discuss issues and questions with the audience. Not all panel members will be present, but we will link to them in Second Life during the panel forum timeslot.

Contributed Papers

An Exploration of Internal Factors Influencing Student Learning of Programming

Angela Carbone[1], John Hurst[4]
Faculty of Information Technology
Monash University, Australia

61 3 9903 1911

Angela.Carbone@infotech.monash.edu.au
John.Hurst@infotech.monash.edu.au

Ian Mitchell[2], Dick Gunstone[3]
Faculty of Education
Monash University, Australia

61 3 9905 2857

Ian.Mitchell@education.monash.edu.au
Dick.Gunstone@education.monash.edu.au

Abstract
This paper explores internal factors influencing student learning of programming. This is based on literature relating to student learning and learning of programming. Two dimensions: *motivation* and *capability* are used as a framework to explore the data gathered from a study of first year undergraduate IT programming students at an Australian University. The authors propose that the exploration conducted in this study is useful in assisting academics with developing tasks to facilitate student motivation and skills to learn programming.

1 Introduction
Literature relating to learning suggests that there are many factors that influence cognitive engagement and the learning process. This includes the individual, the learning environment and the tasks set. The problem of researching learning is that each of these factors is regarded as important, with all having an immediate impact. Consequently, studying any one of them in isolation from the others carries obvious risks. On the other hand, it is difficult, if not impossible to discuss all of them at once. Our research does explore all these factors, but this paper concentrates on one of the major strands, that is, individual factors that influence student learning of programming. This paper is part of a broader study that attempts to pull together information in this field in a comprehensive manner that is integrated into the broader literature on learning.

Copyright © 2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

The assertion made in this paper, is that there exists an individual domain which encapsulates personal factors that influence student learning of programming, and that these factors can be described by a student's motivation towards engaging in the learning activities and his/her capability to complete these activities.

The body of this paper will tease out some of the issues relating to student capability and motivation required to learn to program. Of course, other factors such as the learning environment and the task contribute to the students learning, and once these are fully understood, one can examine the interplay between the set of internal and external influences on student learning of programming. However, the external influences, although explored in the broader study are beyond the scope of this paper.

2 Literature Review
There are many factors that influence cognitive engagement and the learning process. Biggs (1987) groups these factors into student based factors, teaching based factors and the system as a whole. Helme and Clarke (2001) frame these factors affecting cognitive engagement as; the individual, the learning environment and the learning tasks.

The focus of this paper is on developing a framework outlining factors related to the individual, which can be used to promote programming capability. With regard to the individual, Helme and Clarke (2001) state that

> "students need to have both the will (motivation) and the skill (capability) to be successful learners. It is the experience of teachers that students who are motivated to learn and who think carefully about what they are learning develop deeper understanding of the material being covered." (p136).

They also state that "the individual brings to the learning situation numerous characteristics that influence their cognitive engagement. These include: skills, knowledge, dispositions, aspirations, expectations, perceptions, needs, values and goals" (p138) (Helme & Clarke, 2001).

Both studies suggest that personal or individual factors can be described and grouped according to student motivation towards engaging in the learning activities and student capability in completing these activities unspecific to any context. The studies reviewed next therefore consider the types of skill and motivation needed to learn programming. Although these issues are the primary focus of this section, most studies investigated the relationship between academic performance and the students' personal attributes or predisposition factors (Katz et al, 2003; Wiedenbeck, 2005).

One multi-national, multi-institutional study that investigated the programming competency of first year tertiary students found that "many students do not know how to program at the conclusion of their introductory courses" (McCracken et al., 2001, p. 125). Other studies have attributed programming success to factors such as the student's background in maths and science (Byrne & Lyons, 2001; Wilson 2002); (Bergin & Reilly, 2005); learning styles and problem-solving skills (Beaubouef, Lucas, & Howatt, 2001); (Goold & Rimmer, 2000); (Haden, 2006); prior academic experience; self-perception; and specific cognitive skills (Bergin & Reilly, 2005). Other predisposition factors that have been reported in the literature include spatial visualisation skills and map drawing styles, which both had a significant correlation with marks (Simon et al., 2006; Simon et al., 2006; Tolhurst et al., 2006). However, the most frequently mentioned factor for success in programming is previous programming experience (Bunderson & Christensen, 1995; Byrne & Lyons, 2001; Diane Hagan & Markham, 2000; Ramalingam, LaBelle, & Wiedenbeck, 2004; Taylor & Mounfield, 1994; Wilson & Shrock, 2001). These factors are listed to indicate the broad range of issues that researchers have found to influence learning.

In terms of specific skills needed by students to learn programming, fewer studies have been conducted. One comprehensive study by Caspersen (2007) suggests that students need a number of capabilities to become competent programmers. This is because there are a number of steps involved in writing a program. These include: reaching an understanding of the problem to be solved and its solution space; translating the solution into computer terms; testing and debugging then analysing and reflecting on the result (Winslow, 1996). As a consequence, students need many capabilities.

One such capability includes the ability to close track code (Perkins, Hancock, Hobbs, Martin & Simons 1986), yet studies have show that many students are weak at tracing code (Lister et al., 2004) or do not have the willingness to do so (Denny, Luxton-Reilly & Simon, 2008). Others include: the ability to tinker with code effectively, break down a programming task into sub-problems (Perkins, Hancock, Hobbs, Martin and Simons 1986); problem solving skills (de Laet, Slattery, Kuffer, & Sweedy, 2005; de Raadt, Watson, & Toleman, 2006); and the ability to work in a team to solve problems and write programs in collaboration with others (Daigle, Doran, & Pardue, 1996; Hagan, 2004).

Motivation has been found to affect students' learning progress (Helme & Clarke, 2001; Jenkins, 2001). Motivation is an abstract concept and there are a number of theories about it. These include: need and drive theory, which suggests people are motivated by their needs to develop and achieve to their fullest potential and capacities (Maslow, 1962); trait theory, which suggests that motivation is a personality trait of the individual (Kassin, 2003); and other cognitive theories that seek to integrate individual characteristics, job characteristics and organisational characteristics. These theories of motivation are not developed or investigated in this paper. Instead this study determines the type of motivation that students display when working on programming tasks using the concepts intrinsic, extrinsic and achieving motivation (Entwistle, 1998). Intrinsic motivation is derived from a personal interest in the subject. Deci (1975) states that intrinsic motivation is evident when an activity is performed for its own sake and out of interest and curiosity. Extrinsic motivation refers to the desire to complete the course in order to attain some expected reward. Achievement motivation is based on doing well and sometimes performing better than peers (Ryan & Deci, 2000)

The broader study investigates those characteristics of programming tasks that might cause a student's state of motivation to change.

3 Data Collection

This paper reports on a set of individual factors that influence first year undergraduate ICT students learning of programming. The factors are derived from a much larger study that investigated characteristics of programming tasks that influence student learning. These factors are used to construct a framework describing the individual domain that influences student learning of programming. This section reports on the context and design of the study and the participants.

3.1 Context of Study

Data was collected from undergraduate ICT students studying first year programming at Monash University. Two groups of students participated in this study:

- students enrolled in a Bachelor of Computer Science degree studying CSC1011 Introductory Programming (semester 1), and CSC1030 Data Structures and Algorithms, and Computer Systems (semester 2), and
- students enrolled in a Bachelor of Information Management and System (SIMS) degree studying IMS1000 First Year Studio (full year).

CSC1011 provided students with a general introduction to computers and covered basic programming concepts in the C programming language. CSC1030 comprised two components: developing solutions to more complex problems using sophisticated algorithms and data structures; and computer systems, which involves programming using a lower level machine language called MIPS.

IMS1000 was a full-year unit in which students studied Visual Basic programming for six weeks of the 26-week programme. The focus of IMS1000 was on *IT Tools and Technology*. This covered introductory programming concepts, using the Visual Basic programming language for implementation. Other topics were also covered: information management, system analysis, knowledge management and computer systems.

3.2 Project design

Data were gathered from students using the following methods:

i) Semi-structured interviews;

ii) Written descriptions of students' engagement in a task;

Semi-structured interviews with students

The first data collection method used a sample of eight students (four male and four female) from a cohort of 315 Computer Science students who volunteered to participate in the study. The students were guaranteed anonymity, to increase the candidness of interviews and quality of conclusions (Miles & Huberman, 1994), p. 277). All participants were under the age of 20, with programming results ranging from Pass to High Distinction.

Semi-structured *conversational* interviews (Patton, 1990) were used to determine how students perceived the tasks. The eight students were interviewed on a weekly basis for approximately 30 minutes. Students were approached two hours into the laboratory session. The timing of the interview was crucial as current issues and problems that students faced would be at the forefront of their minds and so could easily be captured. The interview was conducted using a conversational style or open approach to questioning. The students were asked to answer some specific questions as part of the interview, for example, "So you are studying programming at university; tell me, what was your first impression of this programming exercise?" Follow-up telephone calls and emails were used to clarify details where necessary. Narrative-analytical accounts (Bassey, 1999) of each student's engagement in a task were transcribed and analysed from the interviews. Once compiled, accounts were returned to the students to be approved and corrected as part of the consent process and to verify that the account was an accurate reflection of what had happened.

The sense during interviews was that participants felt pressure to return to the laboratory to finish the task, which limited the time they had to think and respond thoughtfully. Although some inferences could be made about the sorts of issues affecting students' learning of programming by scrutinising the qualitative aspects of student responses, students did not really elaborate on their learning. This was a problem with the interviews. Students did not elaborate because of time restrictions or because they lacked the vocabulary to describe their engagement in the tasks. Consequently an alternative data collection method was sought to stimulate their thinking about their learning.

Written descriptions of students' engagement in a task;

The second data collection method, which then became the main data collection method, sought data from students via cases in the form of self-reports. Seven CSC1011/CSC1030 students volunteered and all seventy IMS1000 students submitted two written descriptions about their engagement in a task as part of the unit requirement.

Self-reports are a primary source of data in the social sciences. Researchers rely on such reports to learn about individuals' thoughts, feelings and behaviours and to monitor societal trends (Schwarz, 1999). Self-reports are also used to infer cognition, as it is not readily observable (Fredricks, Blumenfeld, & Paris, 2004). Shulman (1997) pioneered practitioner-written casebooks as a professional development tool for teacher educators and staff developers. She describes cases as detailed scenarios written about the real-life experiences of teachers or administrators. Cases reflect reality: they help teachers learn to connect theories and concepts to the complex, idiosyncratic world of practice. The use of cases in this study was to provide students with an opportunity to reflect upon, inquire and analyse their behaviours. The students' stories provided a rich data set.

To help students write self-reports they were invited to attend a workshop titled *Learning how to learn.* The workshop had three broad aims: (1) to make students more aware of their learning; (2) to provide students with a conceptual framework to help them analyse and describe their own learning; and (3) to develop a rapport between the students and the researcher (myself), which would encourage a more thoughtful, reflective exchange of questions and answers. These aims were directed to helping students express and write about their learning, and engagement in tasks. In particular, students were asked to write about their engagement in tasks that left them with powerful impressions of either successful or unsuccessful learning. Their stories also provided insights into their perceptions of other issues that affected their learning of programming.

3.3 Data analysis and reporting

Data was analysed according to the three broad based themes as described in the literature: the individual (internal domain), the learning environment (external domain) and the programming tasks. The internal domain encompasses personal factors unique to the student, the external domain encompasses factors outside the student's control and programming tasks focuses on the characteristics of tasks that influence a students learning of programming. The programming tasks and external domain are not discussed in this paper.

Student quotes were coded into categories that represented the Poor Learning Tendencies and Good

Learning Behaviours (Baird & Mitchell, 1991; Baird & White, 1982b; Baird & Northfield, 1995). Each category was then investigated to determine what other factors, other than the task, might have influenced student learning. These were coded according to *individual* factors and the *learning environment*. Both of these broad categories (*individual* and *learning environment*) were then analysed and recoded into sub-categories. This categorisation is pictorially represented in Figure 1.

The first-level sub-categories for *individua*l factors that influence student learning were divided into *student motivation* and *student capability.* Second-level sub-categories were generated from common themes that emerged from the data itself. As these sub-categories were generated or "grounded" in the data provided by the participants, this part of the analysis used a grounded-theory approach. These sub-categories were expanded and modified as the data was carefully compared against them. The *constant comparative method*[1] (Glaser & Strauss, 1967) was used as the analytical approach to capture common themes across the data.

In the next section when data is reported a specific method of notation is employed. For example, to denote a student's written descriptions of their engagement in tasks the notation *CS.1a* stands for *written description (a) by student 1 in the first year computer science degree studying C programming or MIPS*. If students have written multiple descriptions, each description is labelled *a, b, c* etc consecutively. Any information that may have identified participants has been replaced by a suitable description.

4 Research Findings

The data were analysed to identify

i) causes of shifts in students' motivation, and
ii) a set of skills needed by first year students.

The following sections explore motivation and skills, both technical and generic, which emerged from the data.

4.1 Motivation

This section examines the motivation students displayed when working on programming tasks, using the concepts of intrinsic, extrinsic and achieving motivation.

[1] Constant comparative method is a strategy used to analyse interviews. Four distinct stages are used by Glaser and Strauss (1967). These include: (i) comparing incidents applicable to each category, (ii) integrating categories and their properties, (iii) delimiting the theory, and (iv) writing the theory.

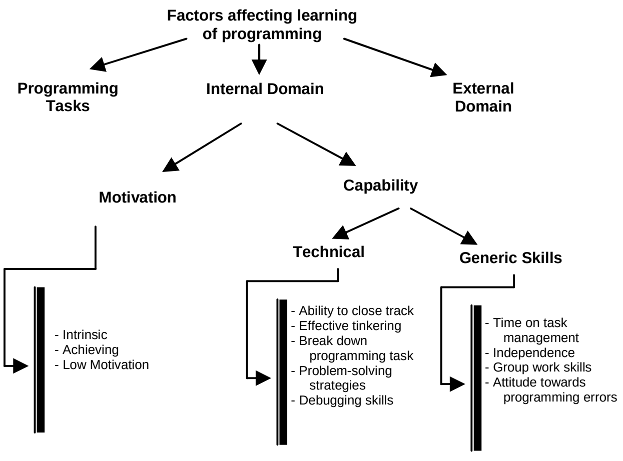


Figure 1 Representation of Internal Domain

4.1.1 *Intrinsic motivation*

In the programming units studied there were students who exhibited intrinsic motivation. These students usually undertook to learn programming in their own time, sometimes prior to the course commencing, working hard at developing their skills. They usually had prior programming experience, providing them with a foundation that enabled them to grasp another programming language easily; they possessed a repertoire of programming skills; and demonstrated a mastery of general programming techniques. Their familiarity with fundamental programming concepts allowed them to concentrate on learning new programming paradigms and language syntax, rather than having to master those basic concepts.

Students displaying intrinsic motivation generally displayed higher capability. These students could apply information presented in lectures to practical problems, and to new situations. They engaged in programming meaningfully and would apply what they had learnt to real life problems, suggesting that a deep learning strategy had been adopted. These students demonstrated a persistence to play around with the code and used alternative resources, not just lecture notes. They played around with the code by making a series of small changes and tests and approached problems methodically and reflectively as compared to others, who approached their work in a more trial-and-error or impulsive fashion. They persevered with compilation and run-time errors, and showed an obvious desire to experiment beyond the requirements of the task. They were also able to overcome difficult challenges.

These students generally worked harder and invested significant amounts of time to complete the task. They often explored different resources and persisted to learn something from the process. Students who adopted such strategies to maximise their understanding repeatedly spoke about a sense of personal achievement and satisfaction when they managed to get their program working.

4.1.2 Achieving motivation

This section reports on students driven by *achieving* motives. Achievement motivation is based on doing well and adopting whatever strategy is needed to secure the best results in the form of the highest marks.

Some students referred to achievement as their main motivator in their written descriptions, either to achieve a pass, or to do as well as they could. The prominence of this type of motivation could have resulted from the nature of the data collected, that is, being based on short-term outcomes, for example, how the task influenced a student's immediate learning. Excerpt CS.4c provides an example of a student driven by the possibility of gaining a high grade and excerpt CS.3b provides an illustrative example of a student wanting to pass.

> The thought of a HD on my end of year report spurred me on. Unfortunately, what one wants and what one gets do not always coincide. Several more attempts at implementing the parts of the questions which I was stuck on resulted in little. Time was running short and I had many other things to do … [Excerpt CS.4c]

> I don't feel like I learnt anything of value, except that MIPS is incredibly frustrating. I knew how to do the prac — I could see it ticking over in my head, but I at least needed to get it working to get a pass grade. That was also frustrating — all of my energies went toward "marks" instead of "learning". It would have been an invaluable prac, but as it was, I walked away with naught but a headache. [Excerpt CS.3b]

These examples illustrate how high levels of motivation are not necessarily associated with high levels of cognitive engagement. This finding is supported by similar observations made by Blumenfeld et al. (1992).

4.1.3 Low motivation

Few students indicated that they had low motivation. The low number could have resulted from the method of data collection. The computer science students volunteered to participate in the project, and none of the participants reported or showed signs of lacking motivation. The students studying Visual Basic were required, as a compulsory part of the unit, to write about their engagement in tasks and it was in these descriptions that the students with poor motivation were revealed. Excerpts VB.16 and VB.63 provide two examples of students explicitly stating that they lacked motivation to engage in a programming task, which was purely based on a lack of interest in the unit.

> As I've foretold earlier, I don't have a great interest in VB that didn't make me strive harder to learn it 100%. [Excerpt VB.16]

> I was too distracted and lazy to concentrate on the learning task at hand. Independent learning requires us, as students to want to learn and not be forced by others to learn. Maybe I still thought I was in high school, and expected the teacher to come around, instructing, "Get to work! Where are you up to? What? You should be up to here by now … " [Excerpt VB.63]

4.1.4 Changes in motivation

An interesting insight that emerged from this study was that students could start off intrinsically motivated to learn, but then, while working on a task, experience a change in their state of motivation. Students could shift from being intrinsically motivated to achieving or low motivation, or move from a state of low motivation to one that was intrinsic or achieving in nature.

In excerpt CS.2a the student originally put extra effort into the task, but since he wasn't rewarded for the extra effort, he adopted a purely achieving learning approach in which his sole aim was to acquire the marks.

> It was the first prac I had for CSC1030 and I put extra effort into it only to find that that wasn't required ... since that time prac, I have just done the pracs with "marks" on my mind instead of trying to make it more efficient and user friendly. [Excerpt CS.2a]

In excerpt CS.5b, the student finds dealing with a bug in the simulator too much of a challenge, and her state of motivation changes. Unfortunately, by the end of the laboratory session, the student had lost the motivation to spend any further time on the task, even after the lecturer had granted her an extension.

> It really burns when someone tells you, you should have done something and you didn't, and you know you should have done it. A bad mark is even worse. And that's what I got for this MIPS prac I had gone to the lecturer just a little upset. Yes, she said, I had encountered bugs in the simulator, and she offered me some more time to complete the prac, but I don't think I could have finished it anyway. All bugs and time wasted aside, once I had walked out of that prac room, I didn't want to know about it. [Excerpt CS.5b]

A more serious concern than motivation changing from an intrinsic state to an achieving one is when motivation shifts from high to low motivation. This is illustrated in excerpts VB.63 and CS.1b. Students can lose interest in what they are doing quickly, and this is usually because they have encountered difficulties in the task, or have no way of proceeding. They can move into state where they no longer wish to continue working on the task, and at times this can lead them to disrupt others, as in the case of excerpt VB.63.

> The learning process was hindered in this instance because once I realised that I was having difficulty completing the task; I decided to spend the studio time talking. This was a big mistake, because the three hours in which I could have attempted to learn something about VB applications turned into something of a gossip session. Reflecting back on the time, I now realise that this is a powerful

> impression of unsuccessful learning as I was also disrupting other students who may have wanted to seriously complete their studio activity. [Excerpt VB.63]

In excerpt CS.1b, the student encounters a problem, and because he is unable to fix it, ends up abandoning the task rather than persisting in order to fix it.

> I had no idea how to fix this problem, and as we only had fifteen minutes left in the prac it didn't look likely that I would be able to solve the problem. So I left. [Excerpt CS.1b]

Just as students can move into a state of low motivation, they can also move into a motivated state. The data suggests that once students acquire the necessary programming skills and develop a familiarity with the programming language, they can become motivated to learn more. So their motivation shifts and feeds on their success. Excerpts VB.37 and VB.39 illustrate this point.

> However, once I got the hang of coding it became very addictive. I could not go to sleep if I didn't get the program running the way it should. Hence I gradually developed an interest that I never thought I would. I also became very interested in coding every exercise in the studios. [Excerpt VB.37]

> The entire impression of successful learning only really took full effect after I had completed the CADAL Quiz and reviewed my results. With persistence and consistent hard work, I had managed to score a perfect mark on the test. Although I did refer back to the notes of the seminar, I was able to remember and consequently learn a lot of the material covered, which I put into practice during the quiz. Even … as delighted and happy as I was it is only now after being presented with such a question, do I seriously realise how that simple activities coordinated and directed my positive impressions and motivation to learn Visual Basic. [Excerpt VB.39]

4.2 Capability

4.2.1 Technical Skills

The technical skills that students lacked are discussed below.

Inability to close track

Students didn't sufficiently track their code to localise problems accurately.

> I started to do it, designed the interface for the VB application, and wrote the codes for the simple ones... When I wanted to use the Do … Until to write the coding, I met the problem, I notice that when I running the program, was not be looped as I wished. So I knew the coding that I wrote maybe wrong, then I rewrote the code more than fve times, but it still got the same problem.
>
> I had no ideas by myself, time gone so quickly, and just left one hour to finish. So I found the tutor and asked him how to solve this problem, and the tutor so kindly and tell me what was wrong for my coding, but he did not help me to rewrite the coding, and gone away. [Excerpt VB.19]

As in the case above, the student attempted to correct the code over five times but each repair was incorrect so the student was unable to isolate the problem and complete the program on his own. As a result, he sought assistance from the tutor who was able to track the problem and advise him on how to proceed. Although students may demonstrate an eagerness to tinker with the code, they fail to track errors, and rely on tutors to do that for them.

Ineffective tinkering

> I had no idea where my program was wrong; all I could do to fix the program was try as many different things I could. [Excerpt CS.1a]

Students often lack the ability to tinker effectively with their code, producing error after error, and adding more and more errors every time they make a change to the code. Instead of rethinking how they are solving the problem and question their understanding they continue to make minor changes to correct the code. These students tinker without sufficient tracking and therefore have little grasp of why the program is behaving like it is.

Inability to breakdown a programming task

Students in this study did not to recognise the need to break a programming problem into parts. Data shows that students only considered breaking down the problem and testing it in parts when the idea is suggested by the demonstrator.

> Help was required. I asked my demonstrator to check over the code, see if they could find any obvious errors, but they just scoffed when they saw the mess in front of them. "Why didn't you break it up into sections?" they asked. "You start with a small piece, test it, see if it works and then move on. There could be a million things wrong here". .My eyes began to burn at the thought of restarting. Instead, I attempted yet again to debug. Hours later, which turned into days later the problem had not shown itself. All I knew was there was a problem with one of the loops, but which one and where? [Excerpt CS.4b]

When students did have to break down problems in nontrivial ways, they often faltered. Perkins et al. (1986) claims that this strategy may be feasible for expert programmers because they have at their disposal a well-developed repertoire of programming plans for different chunks of the programming task. However, for the novice, with a scanty repertoire of programming plans, this often leads to an unworkable breakdown of the problem. (p51)

Lack of problem solving skills

Many students would dive straight into the coding part of programming without a clear understanding on the problem or a systematic strategy to solve it. Students commonly would work out a solution to a problem by trial and error, without a plan, trying to solve their problem haphazardly, and relying totally on feedback from the compiler to correct their errors.

> We usually would work out a problem by trial and error. That would involve responding to the many error messages we would come across and trying to find the problem by looking back at the lecture notes and if possible previous VB exercises. [Excerpt VB.1]

In most cases, the compilation process produces a series of error messages highlighting the incorrect use of the syntax of the language. The compilation errors can be eliminated by an iterative process of modifying the code and recompiling, yet students found this process overwhelming especially when they are confronted with many errors. Once the compilation is successful the student runs their program to see if it executes. Sometimes further errors are encountered during the execution of the program; this may be the result of a logical problem (ie. divide by zero error or the actual logic of the flow of control is incorrect) or runtime problem.

Limited Skills in Debugging

The common practice for students was to type their whole solution, without any testing along the way. In one case, the student's program produced a series of syntax errors during compilation consisting of multiple "bad address" error messages during execution.

> I loaded the whole program to firstly weed out the obvious syntax errors. I then, optimistically, tried to run it. Who doesn't check to see if its going to work first go?
>
> "Bad address at …"
>
> "Bad address at …"
>
> just kept scrolling down the screen. I couldn't see what was causing this and neither could the demonstrator. I spent ages by myself stepping through trying to find this bad address, but I couldn't see what was wrong. [Excerpt CS.5b]

This students, was a typical case, lacking basic debugging strategies and skills in using the debugger to help complete the task.

4.2.2 *Personal Skills*

The personal skills that students lacked are discussed next.

Poor Time Management

Despite efforts to manage time, many students failed to produce a complete working system within the time constraints. They spent all of their time on one aspect of the problem and many were unable to progress until they achieved a fully working, bug free, solution. Initially students wanted to build an understanding of the problem at hand however, once this became an excessively time-consuming activity with limited success, students opted for strategies that limited their learning, instead of re-evaluating their plan of attack, and assessing what they have learnt or whether they should continue.

Students spoke about "running out of time" and/or the pressures of having to work within a restricted timeframe. Students driven by achieving high grades prioritised their activities, making conscious decisions about what programming activities they would or would not do.

> I had little patience to spare at the moment with exams approaching like a mad dog to a piece of meat and ten billion other projects/ assignments/ pracs to complete – none of which were easy, mind you. I decided to do the best I could on this prac, but I would not devote any large amounts of time to problem solving or de-bugging – I just could not afford to at the moment. [Excerpt CS.4c]

To save time, students restricted the type of learning approach they would take. As in the case below, the student saved on time by copying slabs of code directly from the text book.

> After a good fifteen minutes, the underlying sense of it all was not sitting quite nicely, as I would have hoped. The sample code seemed to be making sense, but I was struggling to gain an overall picture. I considered spending more time looking at my notes. But time was running short and I had a prac to finish. In the hope that my actually implementing the code would concrete the concept, I dove straight into it… but hit the ground very quickly. Now I was getting impatient. "That's it", I thought, "I will do this the crude way. I will copy the notes." So that is precisely what I did. [Excerpt CS.4b]

Interviews with students revealed that students who copied slabs of code to complete their task didn't search for a deep understanding, they were happy to apply their attention superficially and succeed regardless.

Independence

A common theme emerging from the data was the tendency of students to seek assistance from friends when they ran into difficulty tackling a task. Assistance was provided by friends who were commonly in a senior year level having more programming experience. Although, friends acted as valuable resource, vital in getting students out of situations when they became stuck, students would rely and act on their friend's suggestions, regardless of whether or not they understood the consequences of their suggestion.

> When my friend told me to write a piece of code, I did it. When the code involved pointers, I leant heavily on his suggestions, making little effort to really figure out their background or implications. There was just not enough time, I told myself. The code was working and that was all I cared about. [Excerpt CS.4a]

Sometimes, students seeking assistance from friends were left feeling rather inadequate, or in an utter state of confusion.

Groupwork skills

The productivity of the team, in part, depends on students' interpersonal skills. The group dynamics can improve via their social interactions (communication, consultation and meetings) yet, there are risks involved when students lack the skills needed to contribute to the team.

> During the group activity, I was a bit shy, didn't participate a lot and I let my group members do all the work. [Excerpt VB.26]

Group members who fail to participate in group discussions inhibit the facilitation of social interactions that are necessary for successful learning.

Attitude towards programming errors

Students showed a variety of attitudes to handling programming errors. It was common for students to deal with errors by either stopping altogether and moving onto the next problem or repeatedly trying something different without reflecting on what had been done. These types of strategies are referred by Perkins et al. (1986) as *extreme stopping* and *extreme moving* respectively, with the students labelled as *stoppers* or *movers*.

As one example, the student in excerpt CS.4b illustrates the case of extreme stopping. The student did not know the answer to the problem and was unwilling to explore it any further, so promptly moved onto to the next task.

> Almost instantly, another problem hit me. How on earth was I suppose to implement that in MIPS code? "Stuff it" was my almost definite answer, and moved promptly onto the next problem of the "a[i]". [Excerpt CS.4b]

In the next case, the student never for a moment stops to reflect on what has been done. The students goes around in circles, retrying approaches that have already proven unworkable, yet instead of dealing with the mistakes and the information they might yield, ignores them and continually moves on to keep themselves busy without pausing to think about what might be causing the problem at hand.

> I was going around in circles. Maybe if I run it again it will work …? I'll just run it another time to see if it will work … I had got to the point where I wasn't doing anything constructive. [Excerpt CS.5b]

Perkins labelled students as stoppers or movers. However, this study revealed that students can behave differently in when working on different tasks. The important point is that students sometimes do not see their error messages as vehicles for insight or learning. When students adopt an *extreme stopping* type behaviour they give up all hope of progressing and are reluctant to make any further changes. This has implications for their motivation. *Extreme moving* should also be criticised as in this case students are not reflecting on the meaning of the errors generated, and/or are failing to interpret the errors correctly, thereby making changes without any systematic plan. At least these students are still motivated to try different things, although not behaving metacognitvely.

5 Discussion and Conclusion

Motivation appears labile, that is, it moves easily and is sensitive to other factors. Reasons that explain why students start with an intrinsic motivation, and then change, are offered by Jadud (2006). Throughout this paper, examples have been presented of how students moved between different states of motivation. Students who were intrinsically motivated to learn could move to an achieving state, depending on the challenges they faced. It appears that to sustain intrinsic motivation there needs to be some success achieved otherwise students may resort to a focus on marks or other signs of external achievement.

In this study there were very few students who spoke about having low motivation. Most students wanted to learn and to understand. Rather, the problem was the skills they lacked to complete the task at hand, which dampened their motivation. The lack of skills influenced motivation in a negative direction and the presence of programming skills appeared to increase motivation. It is the academic's role to exemplify, nurture and facilitate that desire and to move motivation in the "right" direction.

Deficiencies in five technical skills emerged: the inability to closely track code; ineffective tinkering; the inability to breakdown a programming task; lack of problem-solving skills; and limited debugging skills. The generic personal skills that students lacked were time management; working independently; group work skills; and a positive attitude towards dealing with programming errors.

Three of the five technical skills — tinkering, close tracking and debugging — are interdependent. If students can close track their code, localising their problems and understanding the issues they can make informed changes. As a consequence, their modification to the code would not be a tinker but an informed and well-planned action. However, where students localise the problem but do not fully understand it, a change to a code would be considered a tinker, because the effect of the change on the output could not be predicted. At least if students tinker they have not given up altogether, but in order to tinker, students have to want to make changes, that is, they need the right attitude to deal with programming errors and to be motivated to do so. Although motivation plays an important part in preventing students from giving up and feeling defeated, less complex tasks in which students experience successes could encourage them to continue until they solve the problem.

The two other technical skills; breaking down a programming task and problem-solving skills were also interdependent. Students cannot break down large and complex programming tasks if they lack problem-solving skills. Studies into problem solving in programming, which originally started in the mid 1980s, are now being revisited by researchers as this is a prominent concern.

The generic skills that students lacked have also been highlighted. In the university context, the development of generic skills has not been seen as part of the Information and Communication Technology charter. In the primary and secondary school system there have been a range of attempts to teach generic personal skills; they are now part of curriculum reform. However, recently industry and governing bodies (IEEE/ACM, ACS, DEST and employer reports) have recognised the importance of these non-programming skills in programming.

A number of issues related to the social organisation of the task were raised in the individual domain that relate to group work skills, communication, and being a team player. Finally, students' attitudes to dealing with programming errors would seem to be influenced by the challenges they face which arise from the task.

Throughout this paper various issues have been raised, but only some possible solutions have been provided. These issues need further investigation. For example, if tinkering and breaking down tasks is important, then some tasks need to be set at the undergraduate level that have less emphasis on implementation and more on practising these skills. Teaching considerations should include how students are expected to learn these skills. Are these skills to be acquired from tutors who are typically untrained in teaching or should students be expected to learn these skills for themselves? If students could improve their capabilities then learning the key features of the language would be a much faster and more self-directed process. Some of these issues may be beyond the normal curriculum or outside the awareness of educators, yet they raise implications for teaching and for both course and task design. The next phase of the research investigates external influences.

6 References

Baird, J. R., & Mitchell, I. J. (1991). Some theoretical perspectives on learning, teaching, and change. *Journal of Science and Mathematics Education in Southeast Asia, 14*(1), 7-21.

Caspersen, M. (2007) Educating Novices in The Skills of Programming. PhD Dissertation. University of Aarhus, Denmark.

Baird, J. R., & White, R. T. (1982b). Promoting self-control of learning. *Instructional Science, 11*, 227-247.

Baird, R. J., & Northfield, R. J. (1995). *Learning from the PEEL experience*. Melbourne, Australia: The Monash University Printing Services.

Bassey, M. (1999). *Case Study Research in Educational Settings*. Buckingham, United Kingdom: Open University Press.

Beaubouef, T., Lucas, R., & Howatt, J. (2001). The UNLOCK system: Enhancing problem solving skills in CS-1 students. *ACM Special Interest Group Computer Science Education (SIGCSE) Bulletin, 33*(2), 43-46.

Bergin, S., & Reilly, R. (2005). *Programming: factors that influence success.* Paper presented at the Thirty-sixth SIGCSE technical symposium on computer science education, St. Louis, Missouri, USA.

Bunderson, E., & Christensen, M. (1995). An analysis of retention problems for female students in university computer science programs. *Journal of Research on Computing in Education, 28*(1), 1-18.

Byrne, P., & Lyons, G. (2001, 25-27 June). *The effect of student attributes on success in programming.* Paper presented at the Sixth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2001), University of Kent, Canterbury, United Kingdom.

Daigle, R. J., Doran, M. V., & Pardue, J. H. (1996). Integrating collaborative problem solving throughout the curriculum. *Proceedings of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, 28*(1), 237-241

Denny, Luxton-Reilly and Simon, (2008). Evaluating a New Exam Question: Parsons Problems. Presented at the Fourth International Computing Education Workshop, Sydney, Australia, September 2008.

de Laet, M., Slattery, M. C., Kuffer, K., & Sweedy, K. E. (2005). *Computer games and CS education.* Paper presented at the Thirty-sixth SIGSCE Technical Symposium on Computer Science Education, St. Louis, Missouri, USA.

de Raadt, M., Watson, R., & Toleman, M. (2006). *Chicken sexing and novice programmers: Explicit instruction of problem solving strategies.* Paper presented at the Eighth Australasian Computing Conference (ACE 2006), Hobart, Tasmania, Australia.

Deci, E. L. (1975). *Intrinsic Motivation*. New York, USA: Plenum.

Entwistle, N. (1998). Motivation and approaches to learning: Motivation and conceptions of teaching. In S. Brown, S. Armstrong & G. Thompson (Eds.), *Motivating Students*. London, United Kingdom: Kogan Page.

Fredricks, A. J., Blumenfeld, C. P., & Paris, H. A. (2004). School engagement: Potential of the concept, state of the evidence. *Review of Educational Research, 74*(1), 59-109.

Jadud, M. (2006). Methods and Tools for Exploring Novice Compilation Behaviour. *ICER'06*, September 9–10, 2006, Canterbury, United Kingdom.

Glaser, B., & Strauss, A. (1967). *The Discovery of Grounded Theory: Strategies for Qualitative Research*. New York, USA: Aldine.

Goold, A., & Rimmer, R. (2000). Factors affecting performance in first-year computing. *ACM Special Interest Group Computer Science Education (SIGCSE) Bulletin, 32*(2), 39-43.

Haden, P. (2006). *The incredible rainbow spitting chicken: Teaching traditional programming skills through games programming.* Paper presented at the Eighth Australasian Computing Education Conference (ACE 2006), Hobart, Tasmania, Australia.

Hagan, D. (2004). *Employer satisfaction with ICT graduates.* Paper presented at the Sixth Australasian Computing Education Conference (ACE 2004), Dunedin, New Zealand.

Hagan, D., & Markham, S. (2000). Does it help to have some programming experience before beginning a computing degree program? *ACM Special Interest Group Computer Science Education (SIGCSE) Bulletin, 32*(3), 25-28.

Helme, S., & Clarke, D. (2001). Identifying cognitive engagement in mathematics classroom. *Mathematics Education Research Journal, 13,* 133-153.

Kassin, S. (2003). *Psychology*. USA: Prentice Hall.

Katz, S., Aronis, J., Allbritton, D., Wilson, C., Soffa, M. L., (2003). *A Study to Identify Predictors of Achievement in an Introductory Computer Science Course* . Proceedings of the 2003 SIGMIS conference on Computer personnel research., April 10-12, 2003, Philadelphia, Pennsylvania.

Lister R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, E., Sanders, K., Seppälä, O., Simon, B., Thomas, L., (2004) A Multi-National Study of Reading and Tracing Skills in Novice Programmers, *SIGCSE Bulletin*, Volume 36, Issue 4 (December), pp. 119-150.

Maslow, A. (1962). *Towards a psychology of being*. Princeton, New Jersey, USA: Van Nostrand.

McCracken, M., V. Almstrum, D. Diaz, M. Guzdial, D. Hagen, Y. Kolikant, C. Laxer, L. Thomas, I. Utting, T. Wilusz, (2001) *A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students.* SIGCSE Bulletin, 33(4). pp 125-140.

Miles, M. B., & Huberman, M. A. (1994). *Qualitative Data Analysis: A Sourcebook of New Methods* (2nd Edition ed.). Thousand Oaks, California, USA: Sage Publications Inc.

Patton, M. Q. (1990). *Qualitative Evaluation and Research Methods (Second Edition).* Newbury Park, California, USA: Sage Publications.

Ramalingam, V., LaBelle, D., & Wiedenbeck, S. (2004). *Self-efficacy and mental models in learning to program.* Paper presented at the Ninth Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2004), Leeds, United Kingdom.

Ryan, R. M., & Deci, E. L. (2000). Intrinsic and extrinsic motivations: classic definitions and new directions. *Contemporary Educational Psychology, 25*(1), 54-67.

Schwarz, N. (1999). Self-reports: How the questions shape the answers. *American Psychologist, 54*(2), 93-105.

Shulman, J. (1997). Teaching Cases: New approaches to teacher education and staff development. http://www.ed.gov/pubs/triedandtrue/teach.html Accessed 1 May 2006.

Simon, Cutts, Q., Fincher, S., Haden, P., Robbins, A., Sutton, K., et al. (2006). *The ability to articulate strategy as a predictor of programming skill.* Paper presented at the Eighth Australasian Computing Education Conference (ACE 2006), Hobart, Tasmania, Australia.

Simon, Fincher, S., Robbins, A., Baker, B., Box, I., Cutts, Q., et al. (2006). *Predictors of success in a first programming course.* Paper presented at the Eighth Australasian Computing Education Conference (ACE 2006), Hobart, Tasmania, Australia.

Taylor, H., & Mounfield, L. (1994). Exploration of the relationship between prior computing experience and gender on success in college computer science. *Journal of Educational Computing Research, 11*(4), 291-306.

Tolhurst, D., Baker, B., Hamer, J., Box, I., Lister, R., Cutts, Q., et al. (2006). *Do map drawing styles of novice programmers predict success in programming? A multi-national, multi-institutional study.* Paper presented at the Eighth Australasian Computing Education Conference (ACE 2006), Hobart, Tasmania, Australia.

Wiedenbeck, S. (2005). *Factors Affecting the Success of Non-Majors in Learning to Program.* ICER'05, October 1–2, 2005, Seattle, Washington, USA.

Wilson, B. (2002) A Study of Factors Promoting Success in Computer Science Including Gender Differences. *Computer Science Education,* Vol. 12, No. 1-2, pp. 141-164.

Winslow, L. E. (1996, Sept 1996). *Programming pedagogy -- A psychological overview.* Paper presented at the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Atlanta, Georgia, USA.

Intervention Programmes to Recruit Female Computing Students: Why Do the Programme Champions Do It?

Annemieke Craig
School of Information Systems
Deakin University
Pigdons Road, Geelong 3217, Victoria
acraig@deakin.edu.au

Abstract

This paper looks at intervention programmes to improve the representation of female students in computing education and the computer industry. A multiple case study methodology was used to look at major intervention programmes conducted in Australia. One aspect of the research focused on the programme champions; those women from the computing industry, those working within government organisations and those in academia who instigated the programmes. The success of these intervention programmes appears to have been highly dependent upon not only the design of the programme but on the involvement of these strong individuals who were passionate and worked tirelessly to ensure the programme's success. This paper provides an opportunity for the voices of these women to be heard. It describes the champions' own initial involvement with computing which frequently motivated and inspired them to conduct such programmes. The research found that when these types of intervention programmes were conducted by academic staff the work was undervalued compared to when the activities were conducted by staff in industry or in government. The academic environment was often not supportive of academics who conducted intervention programmes for female students.

Keywords: Recruitment, computing students, female.

1 Introduction

Since the turn of the century there have been fewer students studying computing at all levels of secondary and tertiary education in Australia. In 2007 in Victoria for example, there are significantly less students studying computing at senior secondary school level (see Figure 1) than in 2001. Equally in 2007 there is a reduced number of Victorian secondary schools that offer computing education than in 2001 (see Table 1). The demand from industry for qualified computer graduates however is high and a subsequent skills shortage has seen the Australian Government include computer professionals on the Australian Migration Occupations in Demand List (MODL 2008).

Copyright © 2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

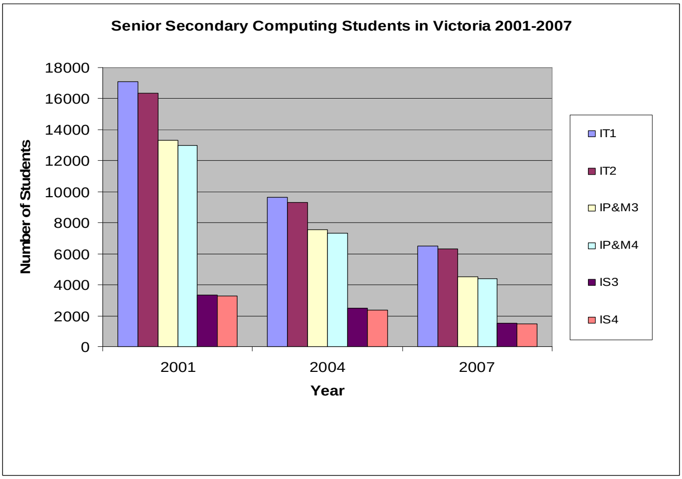


Figure 1: Students satisfactorily completing senior computing subjects 2001-2007 (VCAA, 2008)

Number of Providers	2001	2004	2007
Info. Technology 1	463	409	345
Info. Technology 2	464	413	343
Info. Processing & Management 3	463	405	322
Info. Processing & Management 4	463	404	322
Info. Systems 3	188	174	126
Info. Systems 4	188	174	126

Table 1: Number of providers of senior secondary computing in Victoria 2001-2007 (VCAA, 2008)

Encouraging younger students to consider computing as a future career may help to ease this situation. The literature argues that different strategies are required to engage different groups of students. Even before the downturn in enrolments in computer courses became widespread female students were recognised as an underrepresented group within computer education. The lack of women studying and working in computing is an issue that has been long recognised by academics, the industry and governments in many western countries including Australia (see for example Kay, Lublin, Poiner and Prosser 1989; Clarke 1990; Edwards and Kay 2001; Galpin 2002; Adam, Howcroft and Richardson 2004). Currently female students represent less than 19% of all students Australia wide, enrolled in undergraduate Information Technology courses (DEST, 2008). Consequently numerous intervention programmes specifically aimed at encouraging young females have been designed and implemented (see for example Craig, Fisher, Scollary and Singh 1998; Clayton and Lynch 2002; Goral 2006)

While much has been written about the need for such programmes and the way they may operate little has been heard about the women (and few men) who undertake the task of designing and implementing the programmes. What is their history with the computing discipline and what are the consequences of their involvement with such programmes?

2 Methodology

For this research a collective case study of 14 individual intervention programmes was undertaken. Each of these cases was a concentrated inquiry into a particular intervention programme. Data was collected via detailed document and artefact analysis and by in-depth interviews with the instigator/leader of each of the programmes. Each case was investigated individually to try to understand its complexities (Stake 2000). Within the context of research in the computing discipline, the number of case studies is consistent with other research (Orlikowski and Bardoudi 1991). Myers (2002) queries why more than one case is actually necessary but Miles and Huberman (1994) suggest cross-case analysis enhances generalisability as a multiple-case design will deepen the understanding and ability to explain what has occurred. Herriott and Firestone (1983 as cited in Yin 1994, p. 45) suggest that the evidence from multiple cases is more compelling and therefore the study will be more robust then one of single-case design.

Miles and Huberman (1994) suggest that a multiple-case study requires clear choices about which cases to include within the study. The intervention programmes which became the case studies for this research were selected on the following basis:

- One of the programme's objectives was to increase the number of females who were part of the computing field.
- The programme could be made up of one or more projects however the programme needed to be a sustained activity.
- The principal champion/instigator of the programme was prepared to participate in this research.
- The programme and projects could be completed or be ongoing.
- The programmes were chosen to provide diversity in location and focus.
- The length of time in operation and range of influence of the programme and projects were also considered.

Intervention programmes have been conducted by three different types of entities; educational institutions, government bodies and industry groups. A total of fourteen cases were selected (eight from universities, three from government bodies and three from industry groups). A greater number of case studies were chosen from the university sector than the other sectors due to the proliferation of such activities in this sector.

A total of 19 interviews were conducted. For each case study the programme leader or a major contributor was interviewed. However additional interviews were conducted for five of the case studies either because there were joint programme leaders or another major contributor was available and willing to be interviewed. For four of the case studies two people were interviewed and in one instance a third person was interviewed. Interview times ranged from approximately 60 minutes to 100 minutes in duration. The interview participants are identified by pseudonyms and will be referred to as the programme's champion in further discussion.

3 Focus of the Programmes

The UNDR report (2004, p. xii) advocates that the 'critical starting point for achieving gender in the ICT sector is tertiary level education'. Work by Clayton, Cranston, Crook, Egea, Lynch, Orchard, Robinson and Turner (1993) however, has provided a broader framework which identifies three stages where it is possible to influence the participation by females in computing. The three stages of the framework are as follows:

1. PRE-TERTIARY stage: Where the focus is to encourage females to develop the necessary pre-requisite skills and to enrol in computing courses.
2. TERTIARY stage: Where the focus is to decrease the attrition rates for female students.
3. POST-TERTIARY stage: Where the aim is to equip females with the necessary skills and contacts to obtain positions in the computing profession. (Clayton et al. 1993, p.16):

Eight of the case study entities conducted projects at all three stages (see Table 2), five conducted intervention projects at two stages with only Uni7's focus remaining on just one stage. It is interesting to note that all groups conducted activities at the pre-tertiary stage. Initially involvement by academia and industry commenced with a focus on only one of Clayton's three stages, with a growing awareness of the magnitude of the issues the programmes evolved to incorporate interventions at more stages.

	Pre	***Tertiary***	***Post***
Uni1	☑	✓	
Uni2	☑	☑	☑
Uni3	☑	☑	✓
Uni4	☑	✓	☑
Uni5	☑	☑	
Uni6	☑	☑	✓
Uni7	☑		
Uni8	☑		✓
Gov1	☑		✓
Gov2	☑	✓	☑
Gov3	☑	✓	☑
Ind1	☑		☑
Ind2	☑	☑	☑
Ind3	✓	✓	☑

Legend	
☑	Major Focus
✓	Minor Activity
	No Activity

Table 2: Focus of Intervention Programmes

This paper reports on the findings from this research regarding the champions behind the programmes. All of these champions were women.

4 The Programme Champions

Success of the intervention programmes appears to have been highly-dependent upon not just the design of the programme but on the involvement of strong individuals who were passionate and worked tirelessly to ensure the programme's success. The programme champions were ardent supporters of the cause to encourage more female participation in computing. Their personal involvement with the computing discipline and the intervention programmes are described in the following sections. This involvement was a career barrier for many of the women, though not for all.

4.1 The Journey into Computing for the Champions

While computing for a few of the interviewees was an interest that they had had for a long time, for example Clair received her first Atari at a young age, the majority came to computing by 'accident';

> *I don't have a degree in this area. I haven't got an IT background as such; I fell into it in the telecom's game twelve years ago. My background was definitely not in IT. (Cheryl_Ind1)*
>
> *They said "Do you want to reorganise the library- it's a mess?" and I said (because I liked organising things) "Yeah, I would." So I had the card index system going with everything labelled and David came in one day and he said "Why don't you put that on the computer?" and I said "What?" He said "Why don't you put it on the computer and then people could come in and look it up on the computer VDU." And I said "I don't know anything about computers." And he said "We could do it together." And I said "No, there is no way. I'm not any good at maths." And at the time I was also doing Aran knitting ... and David said …"If you can follow that pattern to create a jumper, you can work computers as it is just another type of code." So I thought okay, so he got my attention with that. I still thought I was going to have to add things up. So we sat down together and he did the programming bit of it but I watched him do it and just created a front-end menu, this was on the green screen, and behind it was my card index system basically which was already, as he pointed out, you already have the logic in the card index system. It was just a case of doing that bit together. So I was hooked. I said "Oh my god. This is how they work?" It demystified it for me and I said "I want to learn more about this". (Bev_Ind1)*
>
> *... it was a complete accident, I fell into it, it bit me and I loved it and that was it and I just stayed. (Lesley_Ind2)*

Ending up in computing without having planned it was not only something that happened to these women themselves but they had also seen this happen with others. Sarah (Uni3) had discovered that a number of her female students 'fell into the [computing] course by chance. Or despite the fact the careers teacher said there is no point in applying for that'. This has led Megan (Uni1) to conclude that;

> *… to a large extent, women's lives in Computing … are governed by serendipity. They don't plan their careers. They look for interesting jobs. Perhaps there really are cultural differences between men and women and no amount of intervention will alter their perceptions of computing careers.*

4.2 Motivation for the Programmes

The original impetus for the intervention programmes that were the subject of this research varied depending on the sector. For the university staff involved, the stimulus for the programmes was often a lack of female students within their own classes. Individual staff realised that there were few, if any, girls within their own computing classes and this led to the development of interventions particularly focused on retaining the current girls or raising awareness amongst secondary girls:

> *We didn't have many women in the course and we wanted to keep the ones that we had. (Alison_Uni5)*
>
> *It was primarily to raise [school girls'] awareness of what a course in IT and working in IT might involve. To get over the stereotypes of it being a nerdy, back-room occupation which did not involve working with people. (Nikki_Uni2)*

Their aim was relatively modest with the desire to increase the number of girls within their own courses a key factor.

> *What we are doing is hopefully going to make a difference and get more girls to consider IT at [Uni3]. (Sarah_Uni3)*

The motivation for the industry groups also emerged from their own circumstances; individual women, seeing the results of a lack of women in computing in their own workplace saw the need to create networking opportunities for women in the industry.

> *We were all in similar positions in an IT company in that we were all starting to hit the glass ceiling ... Hitting glass ceilings and also the decline in numbers of women in the industry. So we started our group as a response to that ... to create a network where [women] could get to hear about opportunities and hear about trends in the industry and what was happening so that you were ahead of the game and not behind it all the time. (Stacey_Ind3)*

The government organisations were a little more pragmatic and they adopted a more holistic view point:

> *I guess everything that frames what we are involved in is around a government policy document … So many different organisations are doing things to engage girls in ICT but we are not getting the full benefit when every man and his*

> *dog is doing something little. It is a bit of a hodge-podge. We wanted to develop future plans for working together. (Clair_Gov1)*

The aim was broader for the government sector with their endeavours intended to change attitudes and practice of primary and secondary school girls, their parents and educators, in relation to computing careers and study opportunities, as well as increasing the engagement of the school girls with computing.

4.3 The Need for Interventions

All of the programme champions were passionate about their commitment to girls and computing initiatives and consider it an important issue to be involved with. Indicative responses of what they saw were the continuing need for such intervention programmes follow;

> *… we all passionately believed that the sort of jobs we were doing were great and being girls, we wanted to share it, and get other girls involved and give [them] opportunities in. (Lesley_Ind2)*
>
> *I think it was always the same need which was to ultimately increase the proportion of women actively working in and shaping what ICT's products and ways of operating were in the community. Ultimately if the community is slightly more than 50% women, you should reflect that in the ICT product services etc, to get relevance. It was driven by everyone's energy and enthusiasm. (Nikki_Uni2)*
>
> *.. every one of them also appreciates that there is a need to encourage other women into the industry. Women I think over a male position is that they have got more of that social conscience and more of that desire to do some community good even though we are overloaded with various things ... I think I have noticed over time that there is more altruism from that point of view and more desire to do something for the career of IT. (Stacey_Ind3)*
>
> *Personally I think that IT benefits enormously from the diversity that women and people from other cultural backgrounds bring. The more viewpoints the better as far as I am concerned and I firmly believe that, I walk it, and I talk it. (Bev_Ind1)*

The views of the programme champions were that with a more diverse workforce in computing, the culture of the computing organisation, and the products it makes, might be quite different;

> *I think that we would probably have less system failures. I think we would have systems that were designed better, more with the user in mind. (Alison_Uni5)*
>
> *For diversity, for every logical reason ... Because it is an untapped resource, it is untapped pool of talent, because girls see things through different eyes to boys. Because Asians see things through different eyes to Australians. We need diversity. (Sarah_Uni3)*
>
> *I have seen a lot of women who are also valuable members of the team because they are able to support each other, the men and the women. They are able to ask different questions about what they are developing, they are able to bring probably the same skills, that is the programming skills but they look at things differently and bring different concepts. They see relationships differently within the application and the abstract aspects. They complement. Often highly-analytical abstract thinkers (say they are men) can be working with highly-analytical abstract women but women are seeing things from different metaphors or different structures so the two come together and create a better application or a better programming outcome. (Stacey_Ind3)*

However there are many assumptions and stereotypes that will need to be broken down before that will occur;

> *I found a job working for an oil and gas company ... When I went there I was introduced around to all of the management team and one chap, Andrew, I'll never let him forget this. The personnel woman was bringing me around and she said "This is Bev, our new IT lady." And Andrew misheard her and said "Finally." and I thought "I wasn't expecting that from the operations manager." And he said "We've finally got a tea lady." (Bev_Ind1)*

It was recognised by the programme champions however that diversity must be managed well if the benefits are to be realised otherwise it may cause conflict and even a decrease in cohesiveness within an organisation.

4.4 Support for the Programmes

Designing and implementing intervention programmes specifically for women in computing does not automatically get support from others in the organisation or community. Support for some intervention programmes was strong and came in the form of the provision of resources and/or an advocate:

> *It was a priority of the Head of School, ... he has set up that supportive environment for them … I think it was certainly embedded into the culture … This is a school where it was considered perfectly reasonable to do an honours thesis on something to do with female participation in IT and women's roles.(Helen_Uni4)*
>
> *Great financial support from the Strategic Initiative Grant. Time release from teaching was purchased, and the freedom to pay for functions for the students from our own budget. (Sarah_Uni3)*
>
> *Thank god we have got the [support of the] university because we have got all the facilities, it is just fantastic. The Executive Dean in the Division of Arts is a supporter. He believes in diversity and he was great at the launch [of the intervention]. (Bev_Ind1)*

Getting support at all was problematic for some with several programme champions having to contend with continually justifying the need for the programmes, or defying opposition to the programmes;

> *It was interesting when I first proposed it which was probably in the late 1990s ...it didn't get department support until I agreed to have a boy's day and a girl's day. (Kerrie_Uni7)*

> *That is the other thing too you have got to convince people that it is a worthwhile thing to do ... It would be really good if instead of having an intervention programme that was run by a couple of interested people, the whole department took it on and said this is important. (Alison_Uni5)*

The industry and government entities comprised mostly of women. The staff from the university sector however were conducting the intervention programmes for girls, through the context of their schools or departments, which were mostly made up of men. Many of the university groups had to counter opposition rather than receiving support:

> *The unintended was the backlash that we got from the boys about why no men's programs. And a degree of defensiveness which I am not sure how much we broke down. (Nikki_Uni2)*

> *We have got a male chauvinist pig in as Head of School who basically denigrates us at any possible chance that he could. Basically criticised our whole courses, the female strong courses, saying are they easier? …. From then it is just a really bad feeling around the place … The [percentage of students who are female] has gotten much less this year. It is the lowest it has ever been … (Fiona_Uni)*

Some of the male students did not support the initiatives either:

> *Some of the male double degree students wanted to come along [to the support community for females] because they thought the other female double degree students had an advantage … There was just a tiny level of snide comments etc and different comments like that. …. They would just say "Your girly stuff" or whatever. It was certainly more than balanced by males who were supportive. But actually thinking about it some were actually supportive in a patronizing way. (Anne_Uni5)*

While the male staff at some of the universities did not support the initiatives this was not the case at all of them:

> *We had a Dean that supported it. We nominated Jack for the equity manager award two years ago for a very good reason; he has always been supportive of women in IT. When we wanted to become a corporate member of WIT, no questions asked. Any of the things that I have tried to do he has always supported. (Jodie_Uni6)*

4.5 Issues with Conducting Interventions

The most common difficulty facing all the women involved with these intervention programmes was the need to invest large amounts of time which detracted from their other responsibilities. Debbie (Gov3) indicated that her involvement with intervention programmes was 'a huge cost in time'. Even the women of the largest of all the case study entities had trouble balancing the time involved, though they were the most successful in terms of being able to share the load;

> *The financial controller I mentioned last night she got married in August last year. I had recruited her on the steering committee. She was really enthusiastic and did a big event in August but she got married in September and now she has realized realistically that her new lifestyle is not going to mean that she has got the time available to do this then. You have to be magnanimous about this and you also have to have a pool of people that is available and up and coming and interested. (Stacey_Ind3)*

The time it takes to be involved and to be able to do things well was by far the most common difficulty faced by all the programme champions with regard to their involvement with girls and computing initiatives, keeping in mind that all were also working full-time and most were mothers and wives as well. Furthermore there was a real concern regarding 'burnout';

> *Again why I haven't really been involved this year is because last year just killed me; it was just so much work… (Fiona_Ind2)*

> *Even when you get government funding for all these programs… We have never had federal government funding but when you get state funding, you are not allowed to pay for the project management and the organisation behind it. Isn't that bloody ridiculous? ... Yeah, every one of those, we have had to put in voluntary… and massive amounts of time. (Lesley_Ind2)*

> *We were starting to do mentoring but it is like everything else, when push comes to shove you just don't have time to do it … I must admit I am a bit naughty I had seen the emails come out [regarding another possible project] and I just thought I haven't got time for it. (Kim_Uni4)*

> *I found when I went back to state A this time, [the women in computing group] has virtually vanished because the key people involved had burnt out, others had moved on … (Helen_Uni4)*

> *Then Josephine decided she was going to put her toe in the water and do [a role model day]. She did and it nearly killed her. I am sure you have heard the story. We thought we should do it, but it nearly killed Josephine and she nearly lost her business. How are we going to cope? … The end result was that it was a mind-blowing experience. … After we did that we thought well it nearly killed us and by this time Josephine was saying "No way am I doing it again ever." (Cheryl_Ind1)*

Nevertheless Josephine, Cheryl and Bev have gone on to organise similar events again;

> *I am coming up to the third one and each time it is like having a child, when we have a child you think "I will never do this again" and then two or three years later here you are pregnant again going through the same issues again. That moment that that child is placed on your tummy is the most special moment you could ever have, you can't replace it with anything. Similarly I think when we had the first [event] and we were there and we were just surrounded by the young girls 13-14 year olds with all the hype and all the energy and all the stress level that got you there and then the day was over and you came back and you thought "What an accomplishment. What a feeling of absolute pure adrenaline pump." … We [recently] played the DVD and it still brings tears to my eyes because I look at the girls and the excitement and I look at the speakers and I look at everything and sometimes I have to pinch myself to realise that we were just a bunch of women who had a passion, who wanted to make a change and a difference, and coming together and say we can do this. All we have is energy and nothing else, and a belief that it is needed. You look at this DVD and think "We created this. This was us. We did it." It makes you feel so good, it really does. That moment! (Cheryl_Ind1)*

The amount of time it takes to be involved with programmes was a key factor in the scaling back, or closure, of intervention programmes in the university sector. Changing environments in universities across Australia has resulted in more students and greater demands upon staff. Kim (Uni4) explained 'we are all being pressured with heavier teaching loads and more commitments'. For the university women the lack of value placed on this work by colleagues was another major difficulty. Therefore when this type of work does not come within a person's job description, is not seen to be important and therefore is rarely supported with adequate funding; it becomes difficult to justify or sustain;

> *We tried to share the costs by also engaging the Computer Science School and with TAFE. The idea was that we would rotate the organisation between those units. The reality was that no-one picked it up. I think eventually we just spat our dummies as it was all extra work, above and beyond, for those who were doing the work. (Nikki_Uni2)*

> *I was actually spending too much time [on intervention programmes]and not keeping up to date with my own area of research which is object oriented systems and development. In the end I needed to make decisions about where most of my time was spent as it can be an all-consuming one and I do tend to get a little obsessive about it. (Anne_Uni5)*

For the women in the university sector their work with girls and computing initiatives was regarded as 'peripheral' to their main job. Legitimacy for participating in 'women and computing' interventions for university staff could come through it being seen as 'recruitment of students' or via a publication record, however it was seen by many colleagues as not real work and not important;

> *I am in a school in a university that doesn't have many women or female staff members and we are not very well regarded by our male staff members in terms of our research, particularly our research of women in IT. … We have been criticised many times that our research is frivolous, it is meaningless and why do we bother? That it is not important. …. 'We do much more important research. We are talking about robots, it is much more important'. In that way it hasn't been so nice and I know I get the impression that we are looked at as the feminist group I guess whereas I don't really think we are feminists as such. I think it is just that we want equity for everyone. I think that is what it is … (Fiona)*

Being labelled as feminists by work colleagues was not an isolated incident;

> *Probably the biggest challenge is that sense that you are a butch feminist pushing an agenda which is not relevant ... You think, boys go and meet a rabid feminist and then meet us! Because it is not where we are coming from. We are not after dominance, we are after equality. We are after the outcome where it is no longer an issue. (Nikki_Uni2)*

Having got a reputation for conducting interventions has also led to an expectation that you could do more. After a very large role model event for secondary girls the state minister suggested that Cheryl's group should run some activities for teachers as well;

> *As we said to the minister it would be nice if we could look after everybody but we can't. It is like George, he wants us to do a showcase … he wants us to do it for boys as well. We can't be everything to everybody because we then won't be successful for anything. (Cheryl_Ind1)*

As mentioned previously by Kerrie at one university the 'Girls in Computing days' were only able to be conducted if there were also 'Boys in Computing days'. At another university it was suggested that the programme champion could also be involved in creating intervention activities for other groups too;

> *In fact I had one lecturer say to me, "Well we have got an issue with non-English speaking backgrounds, why aren't you helping there?" …and my reaction to him was "You're supporting this issue, why don't you do something about it? I haven't got the time to do both" and he backed off very strongly. But it was a shame because I would have liked to have seen some of those issues addressed … Those sorts of issues have to be carried through but one person can't address them all. (Jodie_Uni6)*

4.6 Effect on Own Career

For the women from the university sector being involved in girls and computing initiatives was often not seen as a good career move, with their research output often suffering from the amount of time used up by the intervention programmes. In the university sector a strong research profile is necessary for credibility and therefore promotion;

> *Doing all those sorts of things certainly did not help my research because if I was doing those, I was not doing research. It did, I suppose, get me a reputation for knowing something about it, which in some ways probably did help….. I suppose I have had some interesting invitations to speak because of it, but it did take a lot of time from my research, yes. There were times when I got a bit sick of being "Miss Women-in-Computers", when I actually wanted to get on with doing something else. (Megan_Uni1)*
>
> *It has not just been women-in-computing but student support. I suppose I have been involved in student support when other people have been of doing their PhDs. From that point of view if I had got a PhD earlier I would have finished it. I also would have been promoted earlier. (Jodie_Uni6)*
>
> *It is not an enabler. It is funny with my background [as] a secondary school teacher, education specialist, moving into IT. So I have never really been a solid IT, which is fine because I have never really wanted to be a solid IT. I suppose now I am moving into Information Systems a bit more. I have always looked with a bit of humour at the status. The more technical, the more status, regardless of your education experiences. There are young bucks who have redesigned the whole technical degree with very little educational background. They are allowed to do it because they are the programming specialists. Women in IT is not an enabler.*
>
> *I would never have it as my mainstream research area. Because I don't believe that people take it seriously as a main research area. … I have spoken to people like Sally, she has done a little bit of gender stuff too but she would never do it as her main thing. She is interested in it and sympathetic but she knows that if you want to be accepted in the mainstream that it is a bit of a problem. (Alison_Uni5)*

While the work was not seen as an enabler for an academic career it did have some benefits, as publications could come out of the work and it did make possible contact with influential people in government and industry;

> *I think for me it has given me a lot of opportunity; it has given me access to a lot of people that I probably wouldn't have had access to before. I have had people like the IT Minister, his PA came out and talked to me saying 'What are you doing for girls in IT?' ….we are the people they are coming to. (Fiona_Ind2)*

The effect on the women's careers from industry and government entities was not nearly as negative;

> *No I can't say that it has had negatives. My family suffered because they don't see me but they are so proud of what I am doing, and I am doing something that is making a difference. I am not doing it for me and it is not a selfish motivation that is pushing me forward, it is because I believe in it. (Cheryl_Ind1)*
>
> *The disadvantages are that you tend to get tarred with a brush and people think that that is all you are interested in and that is all you do. If people have heard of me, and obviously not many have but if they have, that would be why and they think that is what you do when you actually have a whole heap of interests and this is just one of them but it is one that I am very passionate about so I don't like getting put on the shelf and "This is who you are." At the same time I also fight to keep doing this because I think it is so important and I don't think it would get picked up if it got dropped. That worries me. … I really enjoy being part of something that is part of the State and being able to have that influence. (Clair_Gov1)*

4.7 Were these Programmes Successful?

How successful were the intervention projects, implemented by the case study entities, from the perspective of the programme champion?

> *Fantastic, incredibly successful. (Debbie_Gov3)*
>
> *The end result was that it was a mind-blowing experience. It was a fantastic effort by all, we just did the unimaginable and pulled it off and we had 1500 girls by the end and we were able to pay our expenses and still manage to do very well. We put together the questionnaire and the feedback just came back from the girls "Fantastic, want it again." … and then as time went on the schools started calling and saying "Are you doing this? We had such a positive experience." … It started building its own momentum. The fact that we have already got 1800 girls registered for the third showcase speaks volumes. (Cheryl_Ind1)*
>
> *Results of the survey show that the intervention project achieved its objective in that it was successful in improving the participants' awareness of the tasks required of a 'typical computing professional'. (Uni7_documents)*

All the interviewees, when asked whether they considered their programmes a success, said 'Yes' though a number of the respondents then qualified their answer;

> *Were the conferences successful? Very much so. I probably would have liked more people at the second one. No, I think they were great. (Kim_Uni4)*
>
> *Yes. We tried to measure it quantitatively and we couldn't. We couldn't relate anything really to the program but our qualitative work tells us that women students perceived that the program was just great and that is enough for me. (Alison_Uni5)*

Almost all of the respondents talked about individual instances of having made a difference in one persons' life and that this was enough to make the programme a success;

Yes. I know that I personally have touched lots of women's lives from that point of view. That satisfies me but does it really justify the amount of time and effort that I have put into it? I don't care. Really you cannot always measure things by the direct and tangible benefits so to me if we supported an activity and it made the difference between no girls going into a course and one girl going in I would be quite satisfied with it, I don't think it is an issue. (Stacey_Ind3)

While nobody described any of the projects or programmes as failures or unsuccessful, not all responses were completely positive:

There is no doubt that the success of the intervention programme depended largely on the personalities and qualities of the senior women students. Naturally some were better at the task than others. (Uni5_documents)

Partial only. I doubt if I move on that it will last independently. The paradox of the majority gender taking no ownership of the issue persists in this Faculty. We have not been able to affect staff perceptions - the plan for a gender unit has failed, there has been no gender awareness training. (Sarah_Uni3)

4.8 Were the Efforts Worthwhile?

Designing and implementing intervention programmes for most of the women was a voluntary task which was not their main 'job'. For many of the women it was a passion which took an incredible amount of time and effort. But these efforts were all worthwhile if it made a difference even to only one participant;

Just seeing the girls faces. There was this one moment, I think we were in [Place X] and there was the robots and stuff and they were programming the robots and this girl was sitting there and she just couldn't get the words out and she was almost shaking because she wanted to say something, she was just so exciting. It was just like "that is awesome", just that moment that is what we are here for. It just makes everything worthwhile. I believe it is making a difference and I think if nothing else it makes me feel good but I think it does actually help and even if it helps one girl that is what matters to me. (Fiona_Ind2)

Almost all the interviewees related stories of having made an impression with some girls that really encouraged them to continue with this work;

We had a set of two girls from a government school who were at our 2004 showcase and we had another two girls from a catholic college who were there. They got up and spoke about the fantastic showcase and how they got so much out of it. "Sincere thanks to the organisers. We would love to see it again and although we are going to be Year 12 next year we are hoping that they will allow us to attend." And I had tears in my eyes. I went afterwards to thank them and I just couldn't. They just really touched my heart. (Cheryl_Ind1)

Even Clair (Gov1), whose principal job did involve girls and ICT activities, considered that the efforts were worthwhile and admitted to spending considerably more time working on these initiatives than what she was paid for;

I think so. I think if I didn't feel that I couldn't come to work. Sometimes it is frustrating but it is more at an anecdotal level. I have seen that kid go off who you know wouldn't have known anything about an IT career and wouldn't have even considered it, going and doing an IT traineeship because she was involved in the girls' computer club and those sorts of things … Seeing the light switch on for teachers as well and just looking at what they can do and that sort of thing … I know in my case anyway it is part of my job and I will do extra hours over and above and it is still quite demanding. (Clair_Gov1)

While there was no doubt in most interviewees' minds that their efforts had been worthwhile there was a disappointment that all the efforts had not brought about more change; that while touching the lives of some girls was really great, the statistics of girls in computing courses were not improving, nor for women in the industry.

One of the most disappointing and frustrating things in writing this has been the realisation that despite the Australian Government's national objective very little has changed in the last ten years and if anything we seem to be going backwards. Participation of women in IT degrees in Australia [is] still nowhere near the objective of 40% … Only 13% of the commencing students in our computing degree programs in 2002 are women. The time effort and money we have put into producing promotional materials and liaison with secondary schools over the years appears to have made little difference. (Uni5_documents)

Despite various initiatives aimed at attracting and retaining more females over many years, the figures [now] are little different. (Gov2_ documents)

4.9 Feelings of Despondency

For all the enthusiasm and commitment for the intervention programmes there was an undertone of despondency amongst the programme champions.

Anne (Uni5) stopped working on intervention projects for a number of years, due to a lack of funding and support and because 'I also didn't think I had any answers anymore. I quite literally thought no this is too big a question to be solving. I just didn't have any answers'. Though she still does not believe she has the answers to the wider problem of females and computing, Anne has begun to be involved in intervention projects again. Jodie (Uni6) also feels that the answer to the 'bigger problem' eludes her:

I have been bashing my head against a brick wall for 20 years, I don't know. It is an attitudinal change, very much, and it is not just the kids. It is the parents and the whole society. I think that

> *there is something wrong in our society for a start that we have such a problem with that dichotomy between males and females and what we do and what we can't do. The reality is so far from the image. (Jodie_Uni6)*

Sarah (Uni3) was concerned about her programme and that it may not ultimately be as successful as she would like:

> *Making no change is a big risk ... Then being ridiculed, "You have had all this money, where are our extra girls? You put your neck on the line … I was very optimistic in 1996 when I started, when I got my teaching fellowship and I thought we were making a difference… I think we are moving forward, yes, in that we are moving away from the 'deficit model' to the 'let us look at IT'. I think we have moved forward in that. We are looking at what IT is. We are asking the questions, the women in Computing. It is only the women asking the questions, very few men take this on board. It is really difficult. When we go to the Women in Computing conference and you preach to the converted. Until blokes see that (WinC Conferences etc) as mainstream it is not going to happen. It is just not going to happen. …. Until it is seen as an issue for all [lasting change won't happen]. (Sarah_Uni3)*

There were concerns expressed about whether, for the individual female who did venture into computing, the discipline would be a good place be:

> *Why would a girl consider choosing computing as a career? If she is happy to go against the mainstream. (Alison_Uni5)*

5 Conclusion

It was the passion, commitment and vast amounts of unpaid time contributed by the programme champions which was the driving force behind the majority of intervention programmes operated by industry, governments and the educational sector.

The success of these programmes was investigated from the perspective of the programme champions as well as via the formal and informal evaluation performed. All the programmes were considered to be successful. Almost all of the programme champions talked about individual instances of having made a difference in one persons' life and that this was enough to make the programme a success. There was disappointment expressed by many of the women however that the incredible time and effort needed to sustain these programmes (most of which was of a voluntary nature) had not brought about more change, with the overall state and national statistics showing little improvement.

The value placed upon the work of designing and implementing intervention programmes varied by sector. When this work was undertaken by academic women in universities the work was frequently undervalued compared to when the activities were conducted by staff in industry or in the government sector. The academic environment was often not supportive of academics who conducted intervention programmes for female students.

With the current poor number of students who are attracted to computer courses it can only be hoped that the need to encourage young boys as well as young girls into computing may see this work more highly valued. The recruitment and retention of computing students right throughout the educational pipeline should not just be left to the women and a few men. If more computer professionals engaged in recruitment activities then the work may not only become more valued but may help stem the fall in enrolments in computer courses.

6 References

ADAM, A., HOWCROFT, D. and RICHARDSON, H. (2004). A decade of neglect: Reflecting on gender and IS. *New Technology, Work and Employment*, pp. 222 - 239.

CLARKE, V. (1990). Sex Differences in Computing Participation: Concerns, Extent, Reasons and Strategies. *Australian Journal of Education*, Vol. 34(1), pp. 52 - 66.

CLAYTON, D., CRANSTON, M., CROOK, M., EGEA, K., LYNCH, T., ORCHARD, R., ROBINSON, P. and TURNER, A. (1993). Strategies to Increase Female Participation in Computing Courses. In *Networking for the Nineties-The Second Women in Computing Conference 1993*. (Ed: FISHER, J.), Victoria University, Melbourne, pp. 14 - 20.

CLAYTON, D. and LYNCH, T. (2002). Ten Years of Strategies to Increase Participation of Women in Computing Programs - The Central Queensland University Experience: 1999-2001. *Inroads SIGCSE Bulletin*, Vol. 34(2), pp. 89 - 93.

CRAIG, A., FISHER, J., SCOLLARY, A. and SINGH, M. (1998). Closing the Gap: Women Education and Information Technology Courses in Australia. *Journal of Systems Software*, pp. 7 - 15.

DEST (2008). Department of Education, Science and Training, Statistical publications, accessed August 22, 2008 from www.dest.gov.au/sectors/higher_education/publications_resources/statistics/publications_higher_education_statistics_collections.htm#studpubs

EDWARDS, J. and KAY, J. (2001). A Sorry Tale-A Study of Women's Participation in IT Higher Education in Australia. *Journal of Research and Practice in Information Technology*, Vol. 33(4), pp. 329 - 335.

GALPIN, V. (2002). Women in Computing Around the World. *Inroads SIGCSE Bulletin*, Vol. 34(2), pp. 94 - 100.

GORAL, C. (2006) *How to Increase Women's Impact on Technology - Changing Culture, Curriculum and Technology*, Anita Borg Institute for Women and Technology.

HERRIOTT, R. and FIRESTONE, W. (1983) Multi-site qualitative policy research: Optimising descriptions and generalizability. *Educational Researcher*, Vol 12(2) pp. 14-19

KAY, J., LUBLIN, J., POINER, G. and PROSSER, M. (1989). Not Even Well Begun: Women in computing courses. *Higher Education*, Vol. 18(5), pp. 511 - 527.

MILES, M. B. and HUBERMAN, A. M. (1994). *An Expanded Sourcebook; Qualitative Data Analysis*

(Second Edition), Thousand Oaks: Sage Publications.

MODL (2008) The Australian Migration Occupations in Demand List available from *www.workingin-australia.com/info/46* released 17th May 2008, accessed August 2008.

MYERS, M. D. (2002). *Qualitative Research Workshop.* Faculty of Computing. Presented at Monash University 2002.

ORLIKOWSKI, W. and BAROUDI, J. (1991). Studying Information Technology in Organizations: Research Approaches and Assumptions. *Information Systems Research*, Vol. 2(1), pp. 1 - 28.

STAKE, R. E. (2000). *Case Studies. In Handbook of Qualitative Research.* (Eds: DENZIN, N. and LINCOLN, Y.) Thousand Oaks, California, Sage Publications, pp. 435 – 454

UNDR: UNITED NATIONS DEVELOPMENT REPORT (2004). Bridging the Gender Digital Divide: Report on Gender and ICT in Central and Eastern Europe and the Commonwealth of Independent State. UNDP Regional Centre for Europe and the CIS / UNIFEM Central and Eastern Europe.

VCAA (2008) Victorian Curriculum and Assessment Authority: *VCE Subject Statistics* Accessed August 2008 from www.vcaa.vic.edu.au/ vce/statistics/subjectstats.html

YIN, R. (1994). *Case Study Research Design and Methods*. London, Sage Publications.

Teaching and Assessing Programming Strategies Explicitly

Michael de Raadt, Richard Watson
Department of Mathematics and Computing
University of Southern Queensland
Toowoomba, Qld, 4350, Australia
{deraadt, rwatson}@usq.edu.au

Mark Toleman
School of Information Systems
University of Southern Queensland
Toowoomba, Qld, 4350, Australia
markt@usq.edu.au

Abstract

This paper describes how programming *strategies* were explicitly instructed and assessed in an introductory programming course and describes the impact of this curricular change. A description is given of how *strategies* were explicitly integrated into teaching materials and assessed in assignments and examinations. Comparisons are made between the outcomes of novices under the new curriculum and results of novices' learning under the previous implicit-only *strategy* curriculum, measured in an earlier study. This comparison shows improvement in novices' *strategy* application under the new curriculum.

Keywords: Strategies, introductory programming, curriculum.

1. Introduction

It is possible to distinguish programming *knowledge* from programming *strategies. Knowledge* involves the declarative nature (syntax and semantics) of a programming language, while *strategies* describe how programming *knowledge* is applied (Davies, 1993). Programming *strategies* involve the application of programming *knowledge* to solve a problem. A literary survey that defines these terms and highlights this distinction is given by Robins, Rountree, & Rountree (2003).

Programming *strategies* can be *plans* as described by Soloway (1985), or *patterns* (Wallingford, 1996), *algorithms*, etc., together with the associated means of incorporating these into a single solution. Soloway suggests programming *knowledge* is not a "stumbling block" (1986, p. 850) for novices and suggests teaching should reach beyond a focus on syntax and target programming *strategies*. Robins *et al* (2003) also suggest that the key to novices becoming effective lies in them learning programming *strategies* rather than acquiring programming *knowledge*.

Another distinction relevant to this study is found between programming *comprehension* (the ability to read and understand the outcomes of an existing piece of code) and *generation* (the ability to create a piece of code that achieves certain outcomes). Whalley et al. contend that "a vital step toward being able to write programs is the capacity to read a piece of code and describe it" (2006, p. 249) meaning that a novice must be able to comprehend a solution (and the *knowledge* and *strategies* within it) before they can generate a solution at the same level of difficulty. According to Brooks (1983), expert and novice programmers can be distinguished by how they undertake *comprehension*. During program *generation* an expert can rely on a tacit body of programming *plans* developed through solving past problems (Soloway, 1986), while novices are traditionally expected to conceive and apply *plans*, with varying degrees of success (Rist, 1991).

Copyright © 2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

The Leeds group (Lister et al., 2004) attempted to isolate the cause of poor novice results measured by the McCracken group (McCracken et al., 2001). The Leeds group reported that many instructors attribute poor results to poor problem-solving ability in novices. The group attempted to create programming questions that required no problem-solving ability to answer. If novices succeeded in the test it would confirm that novices can successfully acquire programming *knowledge* and instructors could put this issue aside and focus their attention on improving *strategy* instruction. If novices failed this test, it would indicate a failure in programming *knowledge*. Results of the Leeds group study, and the BRACElet project (Whalley et al., 2006) that followed, showed that many novices exhibit a fragile programming *knowledge* and very few can demonstrate programming *strategy* understanding in a *comprehension* exercise. It is therefore important to consider both programming *knowledge* and *strategy* together in curricula.

When considering the problems novices are expected to solve in an introductory programming course, de Raadt, Toleman and Watson (2006) use a scale of problems with three levels being "system", "algorithmic" and "sub-algorithmic". The simplest of these is sub-algorithmic level problems, with solutions that do not involve algorithms or system design. Examples of problems of this scale include avoiding division-by-zero, achieving repetition until a sentinel is found, and so on. *Strategies* used to solve problems at this level are particularly relevant to novices in their initial exposure to programming, yet these *strategies* are also a fundamental part of solving problems at any level.

1.1 Previous Work

1.1.1 Initial Study

A previous study (de Raadt, Toleman, & Watson, 2004) found weaknesses in a traditional curriculum used in teaching an introductory programming course to novices where *strategies* were not taught explicitly. Instead, students were expected to learn *strategies* implicitly by seeing examples and solving problems. Students who participated in the study were asked to create a solution to a simple averaging problem. A number of common flaws were detected when students' solutions were scrutinised under Goal/Plan Analysis (Soloway, 1986).

Participating students were not consistently able to:

- initialise sum and/or count variables,
- use a correct looping *strategy* for the given problem,
- guard against events such as division by zero, or
- merge *plans* that should be achieved together.

Students, on average, were only able to demonstrate application of 57% of the *strategies* required for a complete solution. These flaws implied weaknesses in the curriculum being delivered to the students at the time.

1.1.2 Pilot Study

Educational research experiments (Biederman & Shiffrar, 1987; Reber, 1993) have shown that explicit instruction can be more powerful than implicit-only instruction, so it was proposed that programming *strategies* be taught explicitly. A number of attempts have been made to represent sub-algorithmic *strategies* in a form that can be presented to novices; with most recent studies focussing on *patterns* (Muller, Haberman, & Ginat, 2007; Porter & Calder, 2003; Wallingford, 2007). For this study *plans* were chosen as they can be used with multiple paradigms, including the object paradigm. *Plans* can be expressed simply, particularly at a sub-algorithmic level. de Raadt, Toleman and Watson (2006) showed that *plans* suitable for novice instruction at a sub-algorithmic level can be identified in solutions produced by expert programmers. Although *plans* were chosen as a *strategy* representation, the focus of this study is on instruction of *strategies*, and this could be tested with any form of *strategy*.

Before introducing programming *strategies* in a full introductory programming course, a pilot study was undertaken (de Raadt, Toleman, & Watson, 2007). A controlled experiment was conducted that compared two curricula: one including programming *strategies* explicitly and a traditional curriculum that required students to learn *strategies* implicitly. Each curriculum was delivered over a weekend with students who had no programming experience. The experiment showed that it is possible to incorporate *strategies* explicitly into a curriculum. At the end of the weekend, participants were asked to generate solutions to three problems including the averaging problem used in the initial study and two similar problems. Experimental participants, who had been exposed to explicit *strategy* instruction, used *strategies* in their solutions, although no significance was proven as the number of participants was small. After the weekend courses, control and experimental participants were interviewed to probe their understanding of the *strategies* they were exposed to, either implicitly or explicitly. Participants were asked to describe their understanding of the problem statements. They were asked to lead the interviewer through their solution, describing each part. Participants were also asked say if they felt their solution would solve the problem. Participants exposed to explicit *strategy* instruction used terms from a *strategy* vocabulary to describe their solutions and showed greater confidence than those exposed to a traditional curriculum.

After the pilot study *strategies* were introduced into an actual introductory programming course held over a semester. A larger set of programming *strategies* was expressed and incorporated into teaching materials, lectures, formative and summative assessments and the examination.

The main testing approach used to gauge *strategy* application in previous studies was Goal/Plan Analysis (Soloway, 1986). With novices, this approach is limited to analysing solutions generated at or near the end of an introductory programming course. After the pilot study it was proposed that analysis of *strategy* skill should be conducted in more flexible ways throughout the course by taking the ideas inherent in Goal/Plan Analysis and using them to assess student work in assignments and examinations. The following are ways *strategies* were incorporated in assignments and examinations.

- Encouraging students to use particular *strategies* when generating solutions for assignments
- Awarding credit for application of *strategies* in assignment marking criteria
- Using problems that focus on programming *strategies* as part of the final examination
- Analysing examination solutions in a Goal/Plan-Analysis-like manner

Awarding credit for applying *strategies* in assessments was also done to encourage students to value this component of programming and devote more effort to learning it.

1.2 Participants and Setting

Participants in this study were novices studying in a first-year introductory programming course. The course is delivered to students on-campus (approximately 40% of the student cohort) and students studying externally (via distance education, potentially anywhere in the world). On-campus students are expected to attend two one-hour lectures followed later in the week by a one-hour tutorial (in a normal classroom) and a two-hour practical class with computers. External students study independently by reading the same written materials, accessing lectures online, and undertaking tutorial and practical exercises. The course runs twice a year, each year, but this study will focus on the results of three particular cohorts.

Table 1. Cohorts involved in the study

Semester	N	Student Location	Strategies
2003	42	on-campus	implicit-only
2005	36	on-campus, external	explicit
2007	45	on-campus, external	explicit

Table 1 shows which cohorts were the focus of comparisons in this study. The initial study, reported in (de Raadt, Toleman, & Watson, 2004), was conducted 2003 in class with on-campus students only. The later cohorts also included students studying externally as testing was conducted as part of the examination; this also kept participant numbers consistent between comparisons during a period of decline in student numbers. In each cohort, participants included school leavers and mature-aged students. Students were from a range of discipline areas but were primarily IT and Engineering students. The entry standard was consistent throughout the period of study. The mix of students has varied with more non-computing students undertaking the course in later years.

Apart from the inclusion of explicit *strategy* instruction (described in detail in section 0) the curriculum was unchanged between the offerings listed above. The course follows a procedural paradigm using the C programming language teaching topics including functions, data storage, selection, iteration, arrays, I/O and recursion. The instructor was the same in all instances.

1.3 Research Questions

This section is divided into two parts related to two perspectives (integration and impact) taken when conducting this study. This two-perspective structure is mirrored in the Methodology, Results and Discussion sections of this paper.

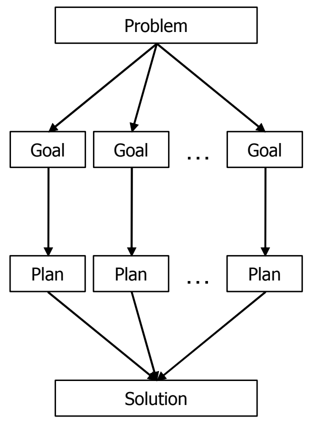
1.3.1 Integration Questions

The first two questions consider the possibility of instructing and assessing programming *strategies* explicitly. Although this was established on a smaller scale in the pilot, it needs to be tested with a complete curriculum in a full-scale introductory programming course.

> ***RQ1.*** *Can instruction of programming strategies be explicitly incorporated into instruction in an actual introductory programming course?*
>
> ***RQ2.*** *Can programming strategy skill be measured as part of the assessment in an actual introductory programming course?*

1.6.1 Design

An expert programmer will take time to properly design a solution. It is tempting to jump to implementation, but often, without a reasonable design, a programmer can waste time correcting a poor implementation and take far longer than if they had spent a small amount of time on design first.



From a *problem statement* a programmer will identify the *goals* that need to be achieved. These goals can usually be found through a careful reading of the problem statement.

STRATEGY

When the goals of the problem have been identified, a programmer can choose appropriate *plans* that satisfy goals. A plan is a small, independent strategy that the programmer has applied in a past solution. During this course we will be covering programming knowledge and also the strategies that you can use to apply this knowledge. Look for the STRATEGY sidebar to differentiate parts of this book that cover strategies.

Once plans have been identified they need to be combined together to form a solution. Plans can be combined together in three possible ways.

- **Abutment**
  Placing the plans one after another in the correct sequence that will solve the problem.
- **Merging**
  Integrating plans so that common parts are performed together
- **Nesting**
  Placing one plan inside another plan

Figure 1. Introduction to *strategies* from the Study Book

1.3.2 Impact Question

The third question relates to the effect of introducing explicit programming *strategies* to novice programmers. This question will be answered by analysing novice performance on assessments in the course and comparing this to the baseline performance described by the initial study (de Raadt, Toleman, & Watson, 2004).

> ***RQ3.*** *What is the impact on novice programmers of incorporating programming strategy explicitly into instruction and assessment?*

2. Integrating Strategies

Over the two-and-a-half-year period between the second half of 2005 and the end of 2007, programming *strategies* have been incorporated into the curriculum of an introductory programming course.

Programming *knowledge* was presented in a similar manner to the traditional curriculum used. *Strategies* are interwoven through the course in an explicit manner. In the beginning of the course the distinction between *knowledge* and *strategies* is presented. Figure 1 shows an initial description of *plans* as *strategies* within a description of the programming process. *Strategies* are a part of the curriculum and testing students' *strategy* skills forms part of the assessment. Students are informed of this at the outset.

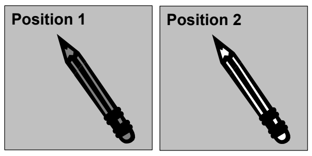
Written materials provided to students include notes for each module of the course and exercises for each week. Students are encouraged to read the written materials before attending or listening to lectures provided online (with audio for external students). The lectures complement the written materials and allow opportunities for questions and further explanations. Each week students are expected to undertake written and computer-based exercises, in tutorials and practicals, to reinforce the material for the week.

The following sub-sections describe how programming *strategies* were explicitly incorporated into written materials,

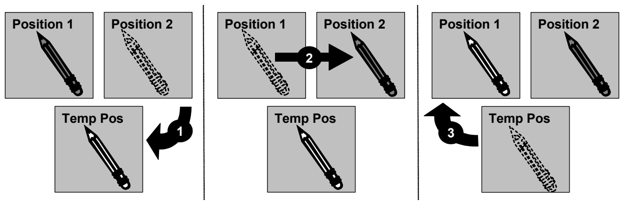
Plan 6. Triangular Swap Plan

This plan requires an understanding of variables and the assignment operator.

Consider how you swap two items. Imagine two pencils in front of you. To swap their positions you would pick up one with one hand, the second with your other hand and then place each in their new positions.



A computer can only perform one action at a time. Now, imagine that you only have one hand; how would you swap the positions of the two pencils now? Keep in mind also that when a variable is assigned a new value, the old value is replaced and cannot be accessed later. Attempting to swap using the above method will result in two copies of the same value.



To achieve a swap a temporary position is needed. One of the pencils could be moved to the temporary position; the second pencil could be moved to its new location; finally the first pencil could be moved from the temporary position to its new position.

Here is an example in the context of a full program.

```
#include <stdio.h>

int main() {
      int firstPosition  = 5; // First position containing value to swap
      int secondPosition = 6; // Second position containing value to swap
      int tempPosition;       // Temporary position for swap

      // Output the numbers after the swap
      printf("Before Swap...\n");
      printf("First: %i, Second: %i\n", firstPosition, secondPosition);

      // Swap the two numbers in a triangular swap
      // 1. Copy the value from the second position to temp
      tempPosition = secondPosition;

      // 2. Copy the value from the first position to the second
      secondPosition = firstPosition;

      // 3. Copy the value from the temp position to the first
      firstPosition = tempPosition;

      // Output the numbers after the swap
      printf("After Swap...\n");
      printf("First: %i, Second: %i\n", firstPosition, secondPosition);
}
```

Here is the output of the above program.

```
Before Swap...
First: 5, Second: 6
After Swap...
First: 6, Second: 5
```

The above results show the values are swapped and not duplicated.

Figure 2. An example of a *plan* from the Strategy Guide

lectures, weekly exercises, assignments and in the course examination.

2.1 The 'Strategy Guide'

The major component of written material provided to novices in the course is referred to as a 'Study Book'. More detail about the Study Book modules is given in section 2.2 below. At the end of the Study Book two appendices are given: one is a syntax guide and the other collects together all the *strategies* that are covered in the course. This 'Strategy Guide' is available online (de Raadt, 2008).

The Strategy Guide begins by defining how *strategies* can be integrated. *Abutment*, *nesting* and *merging* are discussed in this introduction. Each *strategy* is then described as either a *plan* or, in the case of some later *strategies*, as a basic algorithm. An example is given in Figure 2. The programming *knowledge* required to apply each *plan* is stated at the beginning of each *plan* description. Examples and diagrams are provided for most *strategies*. The Strategy Guide forms a resource for novices studying in the course, and possibly after they have completed the course. All *strategies* assessed in assignments and the examination can be found in this guide; students are told this at the beginning of the course and again before the examination. *Strategies* are addressed individually in context within the modules of the Study Book and lectures.

The Strategy Guide contains 18 *strategies* ranging in scale from very simple *plans* such as finding an average, through several sub-algorithmic *plans* such as a triangular swap (see Figure 2 for this example), and on to some algorithmic *strategies* such as sorting. The *strategies* currently in the Strategy Guide are listed below.

1. Average plan
2. Divisibility plan
3. Cycle Position plan
4. Number Decomposition plan
5. Initialisation plan
6. Triangular Swap plan
7. Guarded Exception plans (including Guarded Division plan)
8. Counter-Controlled Loop plan
9. Primed Sentinel-Controlled Loop plan
10. Sum and Count plans
11. Validation plan
12. Min/Max plans
13. Tallying plan
14. Search algorithm
15. Bubble Sort algorithm
16. Command Line Arguments plan
17. File Use plan
18. Recursion plans (single- and multi-branching)

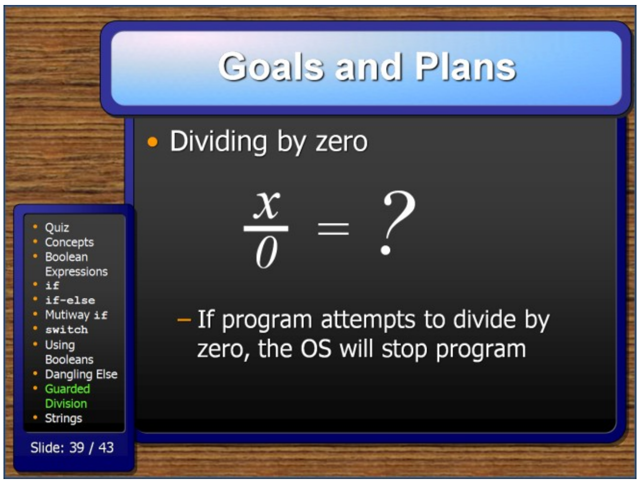
2.2 Explicit Incorporation in Written Notes

Within the 12 modules of the Study Book, programming *strategies* are introduced after presenting the programming *knowledge* applied in each *strategy*. In this context the *strategies* show immediately how the *knowledge* can be applied, which, in its purest sense, is the nature of a *strategy*. This is followed by a code example showing the *plan* applied. For instance, the Triangular Swap plan is shown after students cover *variables* and *assignment* as programming *knowledge* components. This takes place in the third module, covered during the third week of the course. This *plan* is discussed in lectures, reinforced in tutorial and practical exercises and assessed in assignments and in the examination. The Triangular Swap plan appears again when the Bubble Sort Algorithm is presented in a later module of the course. This demonstrates how identifying *strategies* and creating a vocabulary for *strategies* allows instructors to use this vocabulary, and in doing so, reinforce *strategies* when they appear later in the course.

In the Study Book a sidebar down the left is used to visually distinguish parts covering programming *strategy* from other parts of the Study Book.

2.3 Explicit Incorporation in Lectures

During lectures, *strategies* are presented and discussed after relevant programming *knowledge* content had been covered. Lectures are presented in person to a class of on-campus students. The lecture is also recorded and the slides and audio are presented together and placed on the course website.



Goals and Plans

- Dividing by zero

$$\frac{x}{0} = ?$$

– If program attempts to divide by zero, the OS will stop program

Figure 3. Example of a lecture slide showing the Guarded Division plan

The example shown in Figure 3 is one of a number of related slides that discuss the Guarded Division plan. On the left of the slide the outline of the lecture is shown and the current topic, 'Guarded Division', is highlighted. Observe that much of the previous content of the lecture has covered programming *knowledge*. Before a guarded division can be applied, novices must be aware of the `if` statement and the division operator (covered in a previous module). Students are shown how to apply this plan. This *strategy* is reinforced in the tutorial class held later that same week and is assessed in assignments and has been assessed in the examination.

2.4 Strategies in Tutorial and Practical Exercises

Programming is practiced in tutorial and practical classes. Exercises for these classes are listed in the Study Book following the content of each module. Prior to adding *strategy* content explicitly, the following exercise was given as an example.

> *Write a program that will allow the user to enter words. Use the `%s` format sequence in a `scanf()` call to capture each word one at a time. Find the length of each word using `strlen()`. To end the user input, the user will enter the string `"end"`. At the end of the program, output the count of words and the average length of the words.*

This example demonstrates how novices were expected to learn programming *strategies* implicitly in order to solve problems. The problem statement describes what needs to be achieved, but does not suggest how a solution should be constructed, and no strategy to solve the problem had been given in previous instruction.

Computer Exercises

8 Write a program that will allow the user to enter words. Use the `%s` format sequence in a `scanf()` call to capture each word one at a time (this will skip whitespace between words). You don't have to keep the user inputs in memory; you only need to deal with each word one at a time. Create an array with 256 characters for the input word. Set the maximum word size as a constant.

Find the length of each word using `strlen()`. To end the user input, the user will enter the string "end" (you will have to use `strcmp()` to test for this). You will need to include `string.h` to use these functions. Set the sentinel word as a constant.

At the end of the program, output the count of words, the total number of letters and the average length of the words. Be sure to use a sentinel controlled loop and guard the calculation of the average word length. Keep all numeric values as integers.

Your program should work if several words are entered before the sentinel, or if the sentinel is entered as the first input. Test your program by entering "end" as the first word. Try entering more than one word per line of input.

Figure 4. Example exercise requiring the Sentinel-Controlled Loop and Guarded Division plans. Highlighting (added for this figure only) shows *strategy* content

As a contrast, a new version is shown in Figure 4 above. In the new version students are given the same initial requirement with a few programming *knowledge* embellishments (such as the size of an array). Following this, in the third and fourth paragraphs of the problem statement, *strategy* instructions are given. Students are expected to use a Primed Sentinel-Controlled Loop to achieve repetition; this *plan* is named and its use is directed. The students are also reminded to guard the division when calculating the average. At this stage students are expected to know what a sentinel-controlled loop is and how to achieve a guarded division. This problem relies on students possessing a vocabulary that includes the term 'sentinel', which is used to define the value that, when encountered, will stop the repetition.

13. Fill in the blanks in the following code which swaps the values of two character variables and then outputs the variables new values.

```
#include <stdio.h>

int main() {
     char letter1 = 'a'; // First letter
     char letter2 = 'b'; // Second letter
     char temp = '-';    // Temporary position

     // Swap the two letters in a triangular swap
     __________________
     __________________
     __________________

     // Output the letters
     __________________
}
```

Figure 5. Example exercise from Module 3 requiring Triangular Swap plan

The example shown in Figure 5 requires students to apply a Triangular Swap plan to swap two character values. The *plan* name is mentioned explicitly in the code (in a comment) and three blanks imply the use of the triangular swap. Later in the course this *strategy* is used again in an exercise where students write a function that takes two pointers and orders the values to which they point.

Computer Exercise s

6. Copy the *Guarding Division* function example from page 15 that will calculate an average. Add a `main()` function that will call the `average()` function. It should still work when the value passed to `count` is zero.
   6.1 Remove the guarding `if-else` statement so all that remains in the function is the `return` statement. Now test the function sending zero as the value of `count`. When the program is compiled and run, the operating system should shut the program down and display an error.
   6.2 Restore the guard to the function and test that it works correctly again.

Figure 6. Example exercise from Module 5 testing the Division by Zero plan

Figure 6 contains an example of an exercise that asks students to experiment with the Guarded Division plan. This exercise encourages novices to experience the consequences (a program crash) resulting from dividing by zero. Through this, novices will hopefully come to appreciate the necessity of protecting the division with a guard.

Students are deliberately led to practise application of particular *strategies* for these problems in the same way that an instructor might encourage students to use a particular language construct, such as a `for` loop. In the examination, students are expected to apply required *strategies* without being led in this manner.

2.5 Assignment Instructions

As well as being introduced explicitly into instructional materials, programming *strategies* also became assessable in the course. Sections 2.5 to 2.8 describe how programming *strategies* have been included in assignment instructions and marking criteria as well as how examinations have been designed and marked to include testing of strategy-related abilities.

When teaching *strategies* explicitly, the challenge for instructors is to create problems that focus on particular programming *strategies*. Achieving this allows novices to demonstrate specific *strategies* in assignments and the examination.

- In your program, create the following functions.

…

```
void decryptEncryptLine(int shift);
```

- This function will shift alphabetic characters by the amount of the shift. The function performs in the same manner for encryption and decryption. If the shift is a positive amount, this will shift characters forward (encrypt characters) and if negative it will shift them back (decrypt characters).
- The function will input and process each character one at a time until a newline character is detected. Use a **primed sentinel controlled loop**. Do not try to store or process entire lines.

Figure 7. Extract from assignment instructions highlighting the requirement for a specific programming *strategy*

In assignment instructions students are given tasks that require them to apply specific programming *strategies*. Figure 7 above is an extract from an assignment's instructions where students are asked to use a Primed Sentinel-Controlled Loop to input characters entered by a user until the end-of-line is encountered.

2.6 Assignment Marking Criteria

As well as requiring specific *strategies* to be applied in the creation of solutions, the marking schema used to evaluate solutions also explicitly includes references to specific *strategies*.

In the course described here students participate in electronic peer-review as part of each assignment. Marking schema are constructed well in advance and released as part of the assignment instructions. Students are therefore aware of how their submission will be judged before they submit. They can see that they will receive marks for applying specific programming *strategies*. Being involved in peer-review, students are also expected to be able to judge if a peer-student has correctly applied a specific *strategy* where required by a criterion.

…

Check that no variables are declared outside functions. This does not include global constants.

☐ **A Primed Sentinel Controlled Loop is used to process menu options in the `main()` function**

The function should contain a priming input before the loop and a subsequent input at the end of the loop. If the user enters the quit option in the first instance, the loop body should not be entered.

☐ **A Primed Sentinel Controlled Loop is used to gather characters for input until the end of a line in the `decryptEncryptLine()` function**

The function should contain a priming input before the loop and a subsequent input at the end of the loop. If the user enters a blank line, the loop body should not be entered.

☐ **Code is indented consistently and no line is longer than 80 characters**

…

Figure 8. Extract from the marking scheme showing *strategies* are required in the solution for a programming assignment

Criteria relating to programming *strategies* are mixed with other criteria in each marking scheme. Figure 8 is an extract from the marking scheme for the same assignment that was used in the previous section.

2.7 Examination Questions

Questions in the examination are designed to separate ability in *knowledge* from *strategy* and ability in *comprehension* from *generation*. By combining these aspects, four types of question can be defined as shown in Figure 9.

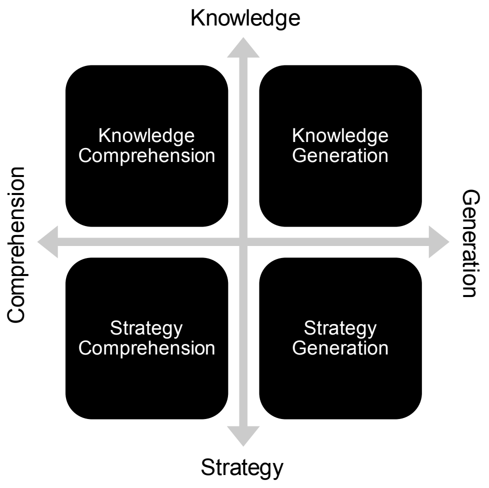


Figure 9. Four types of examination questions based on novice instruction aspects

Targeting questions to one of these four areas is not always simple. Some questions may stray over the boundaries between areas. The focus of the question can be reinforced by criteria used to award marks (see section 2.8).

2.7.1 Knowledge-Comprehension Questions

To test *knowledge* and *comprehension,* an examination question must focus primarily on language syntax skills but not ask the novice to generate any code. The question should test that the student understands an example shown to them, possibly by simulating how the code would be executed. A knowledge-comprehension examination question is shown in Figure 10.

QUESTION 1 (10 marks, 12min)

What will the following output?

```
#include <stdio.h>

int testFunc(int *ptr, int num);

int main() {
      int x=7, y=3, z=5;
      printf("%i %i\n", x, y);
      z = testFunc(&y, x);
      printf("%i %i %i\n", x, y, z);
}

int testFunc(int *ptr, int num) {
      int temp;
      printf("%i %i\n", *ptr, num);
      temp = num;
      num = *ptr;
      *ptr = temp;
      printf("%i %i\n", *ptr, num);
      return num + (*ptr);
}
```

Figure 10. A Knowledge-Comprehension examination question

2.7.2 Knowledge-Generation Questions

Knowledge-generation questions should require novices to generate code but not solve a problem requiring any programming *strategies*. The question should instead prompt the novice to create code that demonstrates their understanding of specific language constructs. An example of such a question is given as **Error! Reference source not found.**.

QUESTION 4 (10 marks, 17min)

Write a `main()` function that input an integer from a user and then use a `switch` statement to respond to the user's input with one of the following outputs:

Where `0` is entered, output `hello`
Where `1` is entered, output `bye`
Where any other value is entered, output `invalid`

Figure 11. A Knowledge-Generation examination question

2.7.3 Strategy-Comprehension Questions

Strategy-comprehension questions are perhaps the most difficult to define. These questions must test the *strategy* potential of a novice without asking them to generate any code. Possible ways to achieve this include the following.

- Asking novices to identify or describe *strategies* used in a given solution
- Asking novices to relate common *strategies* applied across multiple solutions
- Asking novices to identify how a *strategy* has been incorrectly applied in, or is absent from, a solution

In Figure 12 we see an example of a strategy-comprehension question that asks the novice to identify the strategy-related error in the code and state how the error could be corrected. The error can occur when the argument count has a value of zero, which would cause a division by zero. There is no guard to protect against this. To remedy this problem the student should apply a guard against division by zero. The exact 'Guarded Division' terminology is not critical if the novice can express this solution using other words.

QUESTION 5 (5 marks, 18min)

The following function contains a logic error. In a few words, describe what the error is **and** how you would remedy the error. Do not re-write the whole function.

```
int getAverage(int sum, int count) {
      return sum/count;
}
```

Figure 12. A Strategy-Comprehension examination question

2.7.4 Strategy-Generation Questions

Strategy-generation questions are probably what most instructors think of when they write a *generation* question for an examination. Such problems were designed to allow

QUESTION 7 **(20 marks, 24min)**

Write a function, using the following prototype, which will prompt the user and read in a valid positive integer. If the user enters invalid input, or a negative integer, the function will tell them their input was invalid and prompt them to enter another value. The function will repeat this until the user enters a valid input.

```
int getValidPositiveInteger();
```

For your reference, the following lines of code will clear the standard input stream.

```
scanf("%*[^\n]");
scanf("%*c");
```

QUESTION 8 **(20 marks, 24min)**

Write a `main()` function that will read in integers and output their average. Input will be gathered using the `getValidPositiveInteger()` function as described above (do not re-write that function). Stop reading when the value 99999 is entered (this is not to be used as an input).

Figure 13. A Strategy-Generation examination questions

novices to apply specific *strategies* they have learned in the course.

Figure 13 gives an example of two questions that formed a series from the S2, 2007 examination. The first question asks the novice to demonstrate a Validation plan. The Validation plan involves a Sentinel-Controlled Loop plan where a valid input is the sentinel. The second question in Figure 13 is essentially the same classic averaging problem, defined by Soloway (1986), and used in the initial study (de Raadt, Toleman, & Watson, 2004). This question requires novices to apply the following *plans*, each of which is covered explicitly in the course.

- Primed Sentinel-Controlled Loop plan
- Sum plan
- Count plan
- Guarded Division plan
- Average plan
- Output plan

2.8 Marking the use of Strategies in the Examination

When assessing the use of *strategies* in an examination it is critical that the marking scheme does not fall back on syntactical measures. The marking criteria for *strategy* related questions should seek the application of specific *strategies* or *comprehension* of those *strategies*. Strategy-generation questions should target specific *strategies* and the marking scheme for these questions should award marks where the required *strategies* have been applied, rather than for syntactical correctness.

Distinguishing how knowledge-related and strategy-related questions are marked forces a greater focus on particular areas from Figure 9 at the beginning of section 2.7.

3. Methodology

The comparison described in this paper can be considered from two perspectives, which can be related back to the research questions stated earlier:

- to test the possibility of explicitly incorporating and assessing programming strategies in an actual introductory programming course (RQ1 and RQ2); and
- to measure the impact of explicit programming strategy instruction and assessment on novices by comparing results produced under the new curriculum with benchmark measurements from the initial study (RQ3).

The method for achieving these aims is described in the following sub-sections.

3.1 Integration

The first and second research questions (RQ1 and RQ2) raised in section 1.2 consider the possibility of integrating *strategy* content into an actual introductory programming course. The success of this integration, drawing on examples presented earlier, is discussed in section 4.1. Observations are made on student response to the newly incorporated materials and assessment.

3.2 Impact

The third research question (RQ3) seeks to measure impact of the new curriculum relative to curriculum measured in the initial study (de Raadt, Toleman, & Watson, 2004). Students who participated in the initial study had studied using a curriculum that required them to learn *strategies* implicitly. In the initial study students were asked to create a solution to a classic averaging problem. Several *strategy* gaps were detected in student solutions indicating flawed understandings of the required *strategies*. Of particular interest was the lack of application of the Guarded Division plan.

Comparison of performance under the new curriculum with the benchmark performance was achieved through two examination questions. One question was included in the examination that followed the first integration of explicit programming *strategy* instruction in the second half of 2005 and another from an examination at the end of 2007. Results of these two examination question comparisons are shown in section 4.2.

3.2.1 Guarded Division Problem (2005 Examination)

One of the major flaws in novice *strategy* skill, detected in the initial study, was poor use of guarded division. A 2005 examination question shown as Figure 12 (section 2.7.3) is a strategy-comprehension question that targets the Guarded Division plan. This question yields either a correct or incorrect response. Student responses to this question were analysed and compared to application of Guarded Division in the initial study.

3.2.2 Averaging Problem (2007 Examination)

A 2007 examination question shown as Question 8 in Figure 13 (section 2.7.4) was a strategy-generation question that repeated the averaging problem given to novices in the initial study. Solutions to this question were analysed using the same approach as used in the initial study. Eight features were analysed in student solutions: seven *plans*, and the correct merging of *plans*. The presence or absence of each of these features was checked in all attempts. The features measured were as follows.

- Initialisation of a sum variable
- Initialisation of a count variable
- A Sum plan in a Primed Sentinel-Controlled Loop
- A Count plan in a Primed Sentinel-Controlled Loop
- A guard against division by zero
- An Average plan
- An Output plan
- Merging of the Sum and Count plans inside the Primed Sentinel-Controlled Loop

Strategies were judged as being either present or absent in solutions. For more detail on how these features can be identified in a solution, see (de Raadt, Toleman, & Watson, 2006).

The circumstances surrounding the initial testing were slightly different to a final examination. The initial study was conducted under examination-like conditions (students were not permitted to talk to each other or use resource materials), but in tutorial classes during the course. Final examinations are held at the end of the course, giving students more time between exposure and testing of the necessary *plans*. These differences need to be kept in mind when comparing performance between these tests.

3.2.3 Avoiding Bias

Neither of these two specific questions had been used in the course prior to the examinations. The closest problem resembling the averaging problem was the average word length exercise given in practicals and shown in Figure 4 (section 2.4). The course materials covered each of the required *strategies*. Students had opportunities to practice each of the required *strategies*. These *strategies* were not emphasised more than any other *strategies* taught in the course.

In the two examination questions, students are not led to use any specific *strategies*; they are expected to have learned which *strategies* to apply at this stage (during the exam).

4. Results

Results are presented below, again divided by the two perspectives used earlier. First the success of integrating programming *strategies* in an actual introductory programming course is discussed. Specific strategy-related responses elicited under the traditional and new curriculum are then compared.

4.1 Integration

Integrating explicit *strategy* instruction and assessment into an actual introductory programming course was achieved. The examples of curricular materials and assessment items shown in section 0 demonstrate how this was achieved.

Although it is not scientific, some observations can be made. Perhaps the most arduous part of integrating *strategies* explicitly was in conceiving well focused assessment items. It is challenging to create problems that required students to apply specific *plans*, while maintaining interesting problems. Even so, a set of problems was developed to assess *strategy* skill in assignments and examinations.

Students accepted the new instruction as part of the course; no student protested against the inclusion of *strategies* as legitimate content. As each new cohort undertook the new curriculum, they were not aware that it was different to the traditional curriculum that preceded it. Students did not protest against having their *strategy* skills assessed. As mentioned earlier (see section 2.6), assignments involved peer review, so students were being asked to evaluate the work of their peers. Students were asked to complete reviews that required them to judge the presence or absence of *strategies* in the work of their peers.

4.2 Impact

Two specific questions were used to compare *strategy* skill under the previous and new curricula. The questions were drawn from two examinations, one which took place at the end of 2005 after the first instance of the course to include explicit *strategy* instruction, and one in the most recent instance at the end of 2007.

4.2.1 Guarded Division Problem (2005 Examination)

During the initial study a particularly poorly applied *plan* was the Guarded Division plan, with only four students out of 42 applying this *plan*. In the S2 2005 examination, under the new curriculum, the strategy-comprehension question given as Figure 12 (section 2.7.2) was used to specifically target comprehension of the Guarded Division plan after explicit instruction. This question showed a function used to calculate an average; however, there was no guard around the division so it was susceptible to failure if the count of values was zero. Students were asked to identify the flaw and suggest a remedy.

Table 2. Change in Guarded Division application

	Correct	Proportion
Application in generation study before explicit *strategy* instruction	4 of 42	10%
Comprehension in 2005 exam under new curriculum	25 of 36	69%

Results from Table 2 show the poor application of the Guarded Division plan under implicit-only *strategy* instruction and the potential of students to comprehend this *plan* after explicit instruction. After explicit *strategy* instruction, correct answers to the Guarded Division were provided by 25 of 36 students. This indicates that most students had learned and could comprehend the Guarded Division plan, knowing where it should be applied.

Testing *comprehension* of a *strategy* (as in this problem) is not directly comparable to *generation* of that *strategy* (as with the initial study). However, knowing that 69% of students comprehend the Guarded Division plan should be kept in mind when considering the results of a comparison using a generation task in the next subsection.

4.2.2 Averaging Problem (2007 Examination)

During the examination from S2 2007 the questions shown in Figure 13 (section 2.7.4) were used. From this figure Question 8 repeats the averaging problem used in the initial study (de Raadt, Toleman, & Watson, 2004).

Solutions to this problem were analysed under Goal/Plan Analysis, with the same list of *plans* sought. Figure 14 distinguishes results between the initial test, where novices learned programming *strategies* in an implicit-only manner and attempted the problem in class in the second last week of semester, and the examination question under the new curriculum that included programming *strategies* explicitly. Results show consistent improvement in all *plans* except one. The Guarded Division plan is still the most poorly applied *plan*, with only 38% of participants using this *plan* even after explicit instruction; however, this is a significant increase ($\chi^2 \approx 9.47$, $p \approx 0.002$), almost fourfold from the initial study, and this level is higher than the level demonstrated by experts (de Raadt, Toleman, & Watson, 2006). There was also a significant increase in use of the Sentinel-Controlled Count Loop plan ($\chi^2 \approx 4.98$, $p \approx 0.03$).

Figure 15 compares the completeness (use of all expected *plans*) from the initial study and results from the averaging question in an examination under a curriculum with explicit programming *strategies*. Under the new curriculum, the proportion of correct solutions increased from 2% (1 of 42) to 31% (14 of 45) which is a significant increase ($\chi^2 \approx 12.56$, $p \approx 0.0004$). If the most poorly applied *plan*, Guarded Division, is ignored the proportion of complete (and near-complete) answers has increased from 20% (10/42) to 49% (22/45) which is also a significant increase ($\chi^2 \approx 5.88$, $p \approx 0.02$).

Table 3. Improvement between cohorts

Exam	Average Plan Application
Implicit-only (2003)	4.0 of 7 plans (57%)
Explicit (2007)	4.8 of 7 plans (69%)

There was an improvement in the average proportion of application of the seven expected *plans* between the student cohorts. As shown in Table 3, prior to explicit instruction of programming *strategies*, students applied 57% of the expected *plans* on average. With explicit instruction of programming *strategies*, this increased to 69% of the expected *plans* on average. Using a two-sample t-test (one-tailed) there is evidence of a statistically significant improvement between the two cohorts (df=85, $t \approx 1.66$, $p \approx 0.02$).

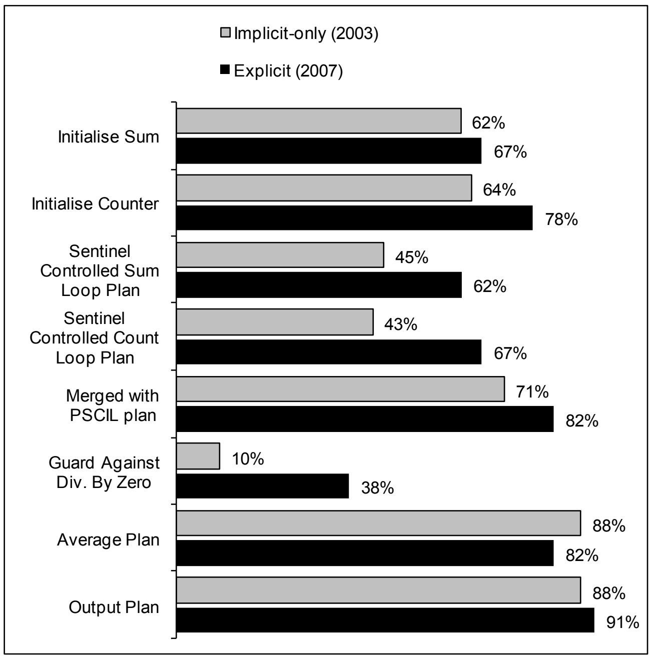


Figure 14. Comparison of *plan* use in the averaging problem between implicit-only and explicit *strategy* instruction

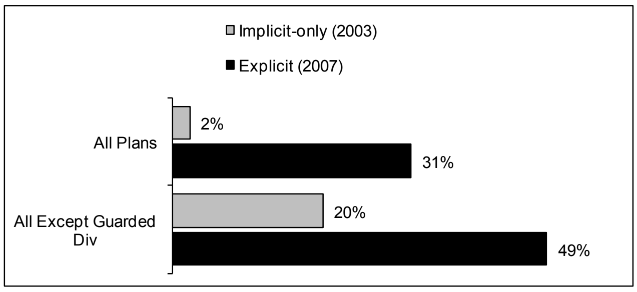


Figure 15. Comparison of complete and near-complete correctness in averaging problem before and after explicit *strategy* instruction

5. Discussion

In this section we use the results from section 4 to answer the research questions posed in section 1.3.

5.1 Integration

> ***RQ1.*** *Can instruction of programming strategies be explicitly incorporated into instruction in an actual introductory programming course?*

While it did take some time and effort to transform a traditional curriculum, adding explicit *strategy* content, this was shown to be possible. The amount of *strategy* content is not necessarily fixed and needs to be further refined. Sharing these *strategies* with other instructors will allow this development. It is useful to reiterate that *strategies* can be used with most imperative and object-oriented languages so they would suit the majority of introductory programming courses, requiring little change for different languages.

> ***RQ2.*** *Can programming strategy skill be measured as part of the assessment in an actual introductory programming course?*

It is possible to measure programming *strategy* ability in novices with tests that address both *comprehension* and *generation*. A number of different forms of assessment have been demonstrated for programming assignments and examinations, providing additional instruments, beyond Goal/Plan Analysis for gauging *strategy* skill. Most assessment methods used in the new curriculum resemble traditional curriculum assessment items, but with careful problem design and objective criteria for evaluation, assessment items can be used to focus testing of *knowledge* and *strategies* independently.

5.2 Impact

> ***RQ3.*** *What is the impact on novice programmers of incorporating programming strategy explicitly into instruction and assessment?*

The results show students' use of *strategies* under a curriculum where *strategies* are covered explicitly is better compared to those results achieved under an implicit instruction curriculum. There is a strong improvement in overall completeness of solutions to the averaging problem tested between the initial study (de Raadt, Toleman, & Watson, 2004) and an examination under the new curriculum. There is a specific improvement in the use of the most poorly applied *strategy*, the Guarded Division plan, although its application is still relatively low.

However, the results shown here are clearly retrospective and do not definitively prove the benefits of explicit *strategy* instruction. The results are consistent and the sample sizes provide confidence in the result. However, with two disparate cohorts separated by four years, student capability and individual differences make it difficult to definitively claim that improvement in this very specific task is attributable to the change of teaching method. There is still a need for a more direct comparison to isolate the impact of such instruction.

6. Conclusions and Future Work

This study has shown that it is possible to instruct and assess programming *strategies*. Teaching programming *strategies* in this way creates a vocabulary that can be used in teaching and assessment, and reused and reinforced after they are presented.

This study has also shown that *strategies* can be a valid part of assessment and can therefore be a valuable part of an introductory programming curriculum that aims to train novice programmers to apply programming *strategies*. The methods of *strategy* skill assessment used can be applied to both *comprehension* and *generation* exercises and conducted throughout a course. Strategy-related questions in examinations can elicit results consistent with questions that assess programming *knowledge* skill. *Strategy* skill testing can also be achieved in regular assignments. With a more precise vocabulary for defining a complete solution to a problem, instructors can avoid vague terms such as 'elegance' and 'connoisseurship' when assessing the work of a novice; instead, instructors can point out what *strategies* are absent or misapplied in novices' solutions.

Students seem to learn and apply programming *strategies* more consistently when they are presented in an explicit manner than when they are learned implicitly. However, further experimentation is required to isolate the effects of this approach on the development of novices.

With a well defined distinction between programming knowledge and strategies in an introductory course, there is potential to investigate programming strategies as possible threshold concepts (Boustedt et al., 2007; Entwistle, 2007).

7. References

Biederman, I., & Shiffrar, M. M. (1987): Sexing Day-Old Chicks: A Case Study and Expert Systems Analysis of a Difficult Perceptual-Learning Task. *Journal of Experimental Psychology: Learning, Memory and Cognition,* **13**(4):640 - 645.

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., & Zander, C. (2007): Threshold concepts in computer science: do they exist and are they useful? *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, Covington, Kentucky, USA 504 - 508, ACM Press.

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

Davies, S. P. (1993): Models and theories of programming strategy. *International Journal of Man-Machine Studies,* **39**(2):237 - 267.

de Raadt, M. (2008) Strategies Reference, http://www.sci.usq.edu.au/staff/deraadt/research/dissertation/Strategies%20Reference.pdf. Accessed November 24 2008.

de Raadt, M., Toleman, M., & Watson, R. (2004): Training strategic problem solvers. *ACM SIGCSE Bulletin,* **36**(2):48 - 51.

de Raadt, M., Toleman, M., & Watson, R. (2006): Chick Sexing and Novice Programmers: Explicit Instruction of Problem Solving Strategies. *Australian Computer Science Communications,* **28**(5):55 - 62.

de Raadt, M., Toleman, M., & Watson, R. (2007): Incorporating Programming Strategies Explicitly into Curricula. *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, Koli, Finland 53 - 64.

Entwistle, N. (2007): Conceptions of Learning and the Experience of Understanding: Thresholds, Contextual Influences, and Knowledge Objects. In S. Vosniadou, A. Baltas & X. Vamvakoussi (Eds.), *Re-Framing the Conceptual Change Approach in Learning and instruction* (pp. 123 - 143): Elsevier, in association with the European Association for Learning and Instruction.

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

McCracken, M., Wilusz, T., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., & Utting, I. (2001): A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin,* **33**(4):125 - 180.

Muller, O., Haberman, B., & Ginat, D. (2007): Pattern-Oriented Instruction and its Influence on Problem Decomposition and Solution Construction. *Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2007)*, Dundee, Scotland.

Porter, R., & Calder, P. (2003): A Pattern-Based Problem-Solving Process for Novice Programmers. *Proceedings of the Fifth Australasian Computing Education Conference (ACE2003)*, Adelaide, Australia 20:231 - 238, Conferences in Research and Practice in Information Technology.

Reber, A. S. (1993): *Implicit Learning and Tacit Knowledge*. New York, USA: Oxford University Press.

Rist, R. S. (1991): Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Human-Computer Interaction,* **6**:1 - 46.

Robins, A., Rountree, J., & Rountree, N. (2003): Learning and Teaching Programming: A Review and Discussion. *Computer Science Education,* **13**(2):137 - 173.

Soloway, E. (1985): From problems to programs via plans: The content and structure of knowledge for introductory LISP programming. *Journal of Educational Computing Research,* **1**(2):157-172.

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

Wallingford, E. (1996): Toward a first course based on object-oriented patterns. *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, Philadelphia, PA USA 27 - 31, ACM Press, New York, NY, USA.

Wallingford, E. (2007) The Elementary Patterns Home Page, http://cns2.uni.edu/~wallingf/patterns/elementary/. Accessed 19th November 2007.

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robins, P., Kumar, P. K. A., & Prasad, C. (2006): An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Proceedings of the Eighth Australasian Computing Education Conference (ACE2006)*, Hobart, Australia 52:243 - 252.

Quality of Student Contributed Questions Using PeerWise

Paul Denny
Computer Science
University of Auckland
Private Bag 92019, Auckland,
New Zealand
paul@cs.auckland.ac.nz

Andrew Luxton-Reilly
Computer Science
University of Auckland
Private Bag 92019, Auckland,
New Zealand
andrew@cs.auckland.ac.nz

Beth Simon
Computer Science and Engineering
University of California, San Diego
La Jolla, CA
USA
bsimon@cs.ucsd.edu

Abstract

PeerWise is an online tool that involves students in the process of creating, sharing, answering and discussing multiple choice questions. Previous work has shown that students voluntarily use the large repository of questions developed by their peers as a source of revision for formal examinations – and activity level correlates with improved exam performance.

In this paper, we investigate the quality of the questions created by students in a large introductory programming course. The ability of students to assess question quality is also examined. We find that students do, very commonly, ask clear questions that are free from error and give the correct answers. Of the few questions we examined that contained errors, in all cases those errors were detected, and corrected by other students. We also report that students are effective judges of question quality, and are willing to use the judgements of their peers to decide which questions to answer. We include several case studies of questions that are representative of the kinds of questions in the repository and provide insight for instructors considering use of PeerWise in their classrooms.

Keywords: PeerWise, MCQ, contributing student, question test bank, peer assessment, self assessment.

1 Introduction

PeerWise is an online tool that enables students to create multiple choice questions with appropriate distracters and an accompanying explanation [1]. When these questions are submitted, they become part of a pool of questions that are shared by the entire learning community. Other students in the same course can use the questions for self-assessment, learn from the explanations, evaluate the quality of the questions they answer (using a 6 point, 0-5 scale), and leave comments on questions.

Copyright © 2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Earlier studies of PeerWise show that students answer many more questions than they are required to, and voluntarily use the PeerWise system during the examination study period as a revision resource [2]. Additionally, students who engaged more actively with PeerWise performed better than their less active peers on a final exam [3].

Since students use and seem to benefit from the content generated by other students, we wanted to investigate the quality of the questions that were being produced. Specifically, we wanted to know “Were the questions created by students of high quality?”

The most important definition of quality for us, as instructors, is whether the questions are an effective and efficient tool for helping students learn what they need to know for a course – and specifically for a portion of a written exam featuring multiple choice questions. The 0-5 rating scale in PeerWise was designed to allow students to express something similar. Assuming they give higher ratings to questions they find most valuable for revision and/or learning, these ratings will be a somewhat subjective measure of overall quality.

In a review of the literature on peer review, Topping [6] reported that almost three-quarters of the studies that compared grades assigned by students with the grades assigned by teachers showed a high degree of consistency between student and teacher assigned grades. Sitthiworachart and Joy [5] reported that grades assigned by Computer Science students engaged in peer review activities had a significant and substantial correlation with the grades assigned by tutors. Although it is clear that students are capable of evaluating the quality of work produced by their peers, we wanted to investigate whether the ratings assigned by students using PeerWise were consistent with the quality assessment of staff. To compare our evaluation of quality with the overall ratings assigned by students using the rating system supported by PeerWise, we rated a sample of questions on the same 0-5 scale.

We found that our subjective ratings did not answer questions we had about more objective characteristics of the student created questions. To be certain that students were not wasting their time or being frustrated by “poor” questions, we wanted to determine whether the great majority of questions were “good” in that they were clearly worded, were free of errors, had a reasonable set of distracters, and provided good explanations. To investigate this, we used a simple rubric for classifying questions according to these characteristics.

Finally, we were interested in studying the quality of the questions as it relates to their representativeness for multiple choice style examinations. Here we categorise the structure of the contributed questions in five ways: (a) is there code in the stem, (b) is there code in the answer choices, (c) is there code in both the stem and answer choices, (d) did the question require computing or matching output, and (e) did the question require identification of an error. In previous work, the Leeds group [4] reported on a large study of questions involving code in the stem only (fixed code problems) or code both in the stem and the answer choices (skeleton code questions). In that study, students scored lower on skeleton code questions than fixed code questions. We were interested to see if students generated questions with similar characteristics. Specifically, we were interested if student generated questions would be skewed towards the types of fixed code questions students answered more correctly in the Leeds study.

In this paper we seek to answer these questions. Essentially we want to know how capable students are of creating questions to be used as a learning resource, what the characteristics of those questions are, and how capable students are of recognizing high quality questions.

2 Methodology

This study is based on a repository of 617 questions collected from a standard Java based CS1 course taught in the first semester of 2008 at the University of Auckland. There were 407 students who attempted the final exam for this course, and 366 of these students had contributed at least one question to PeerWise during the semester. These questions were answered a total of 11,189 times.

Each time a question is answered, the student is given an opportunity to rate the question on a 6 point scale, from 0 to 5. This allows the student to apply their own judgements and make a subjective measure of the overall quality of the question. Figure 1 plots the number of responses each question received against the average rating assigned to that question by the students who answered it.

2.1 Overall quality ratings

Our methodology was driven by an observation we made from Figure 1. It is clear from the figure that poorly rated questions do not receive a large number of responses. Given a large number of questions from which to select, students prefer to choose those that are more highly rated. This is supported in PeerWise as students are able to list all of their unanswered questions in decreasing order of rating. This provides evidence that students do value the ratings that are assigned to questions by their peers, as they use those ratings to help determine how to spend their time revising.

This observation motivated the first aspect of our study. Given that students clearly value the ratings of their peers, we wanted to measure the accuracy of these ratings. We examined a sample of 61 questions, approximately 10% of the repository, by selecting every 10th question in chronological order. For each question in the sample, each author of this paper assigned a subjective quality score to the question using the same 6 point scale (0 to 5) that the students used. In this paper, we will refer to these ratings as "instructor ratings". Our only agreement was that we would define

- 0 “complete nonsense”
- 1 “pretty poor”
- 2 “some merit, but not that useful”
- 3 “average question, learned something”
- 4 “good question, not quite exam quality”
- 5 “good enough for an exam (or almost)”

We expected differences in our application of this scheme, likely based on content areas stressed in the questions. We report on each instructor’s ratings in comparison with each other and with student ratings.

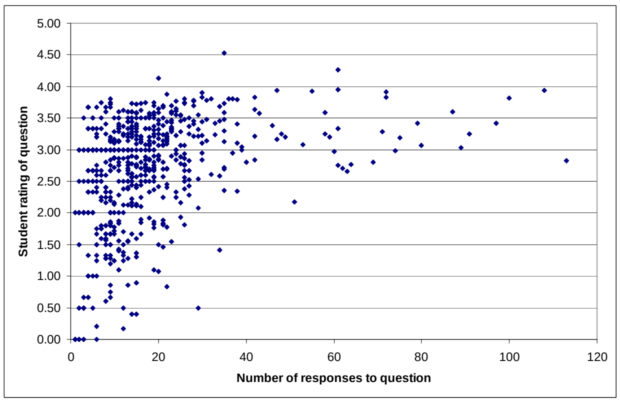


Figure 1: The number of ratings and the number of responses for each of the 617 student questions

2.2 Objective quality ratings

Our overall quality ratings measure the quality of a question as a whole. There are also certain individual characteristics of the questions that we would like to evaluate. The goal here is to improve our understanding of these different elements in order to identify common problems. Ultimately, such an understanding may allow us to help guide students in designing better questions, which would in turn lead to questions receiving higher overall quality ratings in the future.

The rubric we defined and applied to the sample of 61 questions (defined in Section 2.1) is described in Table 1.

One of the elements of this rubric assesses the "correctness" of a question. The correctness of student created questions is likely to be of great importance to instructors. A repository of questions in which a high proportion are not correct would be of little value and it could be argued may even be misleading to students.

2.3 Quality in relation to exams

The nature of PeerWise specifically targets students in preparing and revising for formal examinations. We therefore wanted to evaluate the repository of student produced questions in relation to various “real” exam questions. Previously, the Leeds study [4] had identified two types of problems typically found in CS1 examinations: fixed code problems (which had code only in the question stem) and skeleton code questions (which had code in both the stem and the answer choices).

Table 1. Objective Quality Rubric

Clarity of question	0: No, had something that made it hard to understand 1: Yes, could understand what was being asked
Error free question	0: No, it contained minor errors, including grammatical errors in the stem or minor syntax errors in code presented in the question 1: Yes
Distracters feasible	0: Less than 2 distracters feasible (of correct type, and could be justified) 1: At least 2 (but not all) distracters feasible 2: All distracters feasible (note: not necessarily "a perfect set" of distracters, such as one might devise for an exam)
Explanation	0: Poor or missing 1: Good, explained why the correct answer was correct 2: Explained not only the correct answer, but included some discussion of common mistakes/misconceptions or explicit analysis of distracters
Correctness	0: Answer indicated by the author of the question was not correct 1: Answer indicated by the author of the question was correct

Analysis of the student produced questions led us to define a more detailed set of structures common to questions including: code appearing in the question stem, code appearing in the alternatives, code appearing in both the stem and alternatives, questions requiring computing or matching output of code, and questions addressing errors, such as syntax errors. We coded all n = 617 questions from PeerWise, coded the 12 Leeds study questions, and coded all multiple choice questions from all 10 formal exams and tests since 2004 from the University of Auckland's CS1 course.

3 Results

Here we present the results of the analyses described in the corresponding sections of the methodology.

3.1 Do students' ideas of quality match those of staff?

As evidenced in Section 2.1, students do value the quality ratings assigned to questions by their peers – those questions with the highest response rates were rated highly by students. Given that this is the case, the accuracy of these ratings is of some importance. To measure the "accuracy" of the student ratings, we compared them with the staff ratings of the same questions. For a meaningful comparison, we considered only those questions (42 of the 61 in our sample) that had at least 5 student ratings.

Two of the authors teach this course at the institution on a regular basis (and one taught the term these student questions were gathered). The other author regularly teaches a quite similar course, but in another educational and cultural context.

Table 2 shows the averages and standard deviations for the three authors' codings. Table 3 reports the correlation between each author's coding and the student ranks and also the pairwise correlation between the three author's codes.

Table 2. Differences in Overall Staff Quality Assessment

Staff Coder	Average Rating	Standard Deviation
Coder A	2.8	1.2
Coder B	2.1	0.9
Coder C	3.4	1.2

Table 3. Correlation between Staff Coders and Students

Students	0.58	0.22	0.52	1
C	0.52	0.47	1	
B	0.42	1		
A	1			
	A	B	C	Students

Figure 2 shows a plot of the average instructor rating and the average student rating for each of these 42 questions using the same 0-5 scale. The correlation between the average instructor and student ratings is 0.54, and a positive trend is visible in the chart.

It appears as though students not only value the ratings that have been assigned by their peers, but these ratings are also reasonably effective at assessing the overall quality of a question. When assigning our overall quality ratings, we took into account the clarity and correctness of the question, the plausibility of the distracters and the quality of the explanation. It is likely that students have taken some of these elements into account as well, and it would be interesting to know which of them the students regard as most important.

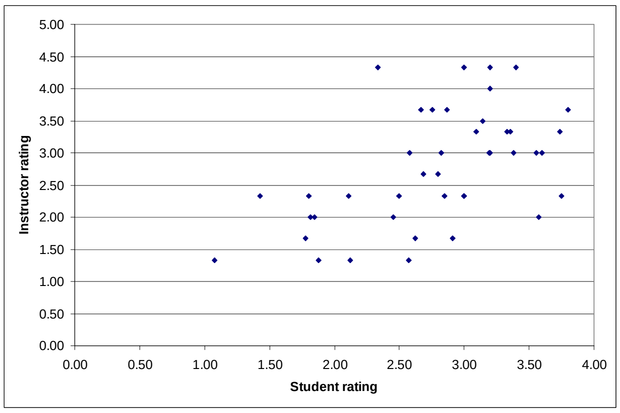


Figure 2: Comparison of the instructor and student ratings, on a 0-5 scale, for 42 questions from our examined sample with at least 5 student ratings

3.2 Objective Quality

Table 4 summarises our classifications of the 61 questions in our sample according to the objective quality rubric presented in Table 1.

We can see that almost all questions were clear and that 80% were error free in the question wording and code (not counting the cases where questions were seeking to have students find errors). Somewhat surprisingly 87% of questions gave distracters that were all feasible. In our definition of feasible, we applied a metric which meant that the answer could be possible from the code – even if it wasn't the "perfect" or even thoughtfully chosen distracter.

Table 4. Object quality classifications

Category	Option	Percentage
Clarity of question	0: No	6.6%
	1: Yes	93.4%
Error free question	0: No	19.7%
	1: Yes	80.3%
Distracters feasible	0: < 2	3.3%
	1: >= 2	9.8%
	2: All	86.9%
Explanation	0: Poor	42.6%
	1: Good	32.8%
	2: Good+	24.6%
Correctness	0: No	11.5%
	1: Yes	88.5%

Explanations were an issue of great interest to the authors – as we hoped they were for the students. We would like students to be able to learn directly from the explanation provided in PeerWise – and certainly we would like explanations not to be incorrect – which would certainly lead to frustration among the students. Unfortunately we ranked 43% of the explanations as poor – they either did not explain the answer clearly or were missing. 33% of the explanations were good in that they explained clearly why the correct answer was correct.

One of the benefits of multiple choice style questions is that students can be directly confronted with misconceptions or alternatives that may be as or more instructive than a simple comparison of the correct answer. We were pleased that 25% of the questions not only featured explanations of the correct answer but specifically discussed either the reasons the various distracters were wrong or in some way addressed a common misconception that may have affected students in answering the question.

3.3 Quality in Relation to Exams

Another objective measure of quality is to describe the structure of the question. Particularly, this analysis draws on the work of the Leeds Working Group which gave a common set of CS1 exam questions to students around the world (n = 556). Those questions fell into two basic categories. First are questions with code in the question stem – called "fixed code" questions in the Leeds study. These problems required reading and tracing of the code – in the Leeds study the answers were either of *int* type or *array* type and asked "what's the value of variable <x> after the code is executed." The Leeds study found that performance on fixed code problems was generally similar and well differentiated student performance.

The second type of question in the Leeds study was a skeleton-code type where code in the multiple choice answers was required to fill in some code in the stem to produce a program meeting certain output requirements. In the Leeds study, fixed code questions were notably easier for students (answered correctly by 68% of students) than skeleton code questions which were correctly answered by 53% of students

We wondered if students' greater ease at answering fixed code questions might translate into a greater propensity to write fixed code questions. The student created questions considered in this study show a greater variety of types than those in the Leeds study, so a strict analysis of fixed code and skeleton code questions would be inappropriate. We broke down question style into 5 more basic components (shown in Table 5), which can be combined to describe the Leeds question styles. We then applied these simple component analyses to both the PeerWise questions and the Leeds questions. It should be noted that none of the "fixed code" Leeds questions actually asked students to identify the output of the code after execution – instead they asked for the value that would be in a particular variable. The 33% (4 questions) of Leeds questions that involved output were in skeleton code questions where skeleton code was filled in to match a desired, described output.

Among student-produced questions, we see that the majority fall in the category of having only code in the stem. Almost all of these are questions that follow the form "what is the output of this code". Approximately 20% of the questions had code in the answer choices, and only 7% had code in both the questions stem and answer choices. Additionally, we note that most of the questions which had code in them involved finding or matching output in some way. In problems where there was code in the answer choices, a common question format might be to ask which of the following pieces of code outputs a certain value. In questions with code in both the stem and the answer choices, a common thing to ask is which of the following "for" loops will produce the same output as this "while" loop.

Table 5. Objective Question Quality by Style

Question Style	Leeds Study	PeerWise	AU Exam
Code in question stem	100%	82%	82%
Code in answer choices	42%	20%	33%
Code in both stem and choices	42%	7%	17%
Find or match output	33%	78%	55%
Identify error	0%	13%	13%

In addition, we can also compare the student created questions to the structure of the questions traditionally given on exams and tests of the CS1 course in our study. In total, 191 questions from ten exams and tests spanning the last four years were analysed.

The proportion of questions with code in the stem and questions which require the identification of an error appear in identical proportions in the 617 student questions and 191 formal test and exam questions. The students wrote a greater proportion of questions that involved finding or matching output, which is likely to be explained by the fact that this style of question is easier to write.

4 Discussion

The correlation between student judgement of quality and instructor judgement of quality (shown in Table 3) gives us confidence that students can distinguish between good questions and poor questions.

4.1 Question Case Studies

In this section, we present four case studies of questions that are representative of several of the kinds of questions in the repository. These case studies are drawn from the entire repository.

Good questions

To highlight the quality of the student generated content, we present here one example of a good question which received an excellent rating from students. There were 75 questions in the repository that had been rated more than 30 times, and of these, six questions had an average rating greater than 3.9. These six questions were contributed by five different students. The question we present here was answered more times than the others (108 responses), and was contributed by the student with two of the top six questions. The distracters were good, receiving a fairly even spread of responses, and the explanation was very detailed and clear.

Which of the following loops could you use to cycle through all elements of the following array without going out of bounds and causing the program to crash?

```
int[] array = new int[15];
```

Assume that in all cases, array[i] will be used without any other math operating inside the square brackets (dereferencing operator).

Each of the distracters for this question are listed below. The parentheses denote how many students selected each distracter.

A (selected by 5 students):
```
int i = 0;
while (i <= array.length) {
   i++;
   // code
}
```

B (selected by 22 students):
```
for (int i = 0; i <= array.length; i ++) {
   // code
}
```

C (the correct alternative, selected by 46 students):
```
for (int i = array.length - 1; i >= 0; i--) {
   // code
}
```

D (selected by 17 students):
```
for (int i = array.length - 1; i > 0; i--) {
   // code
}
```

E (selected by 18 students):
```
int i = 1;
while ( i < array.length) {
   i++
   // code
}
```

The explanation provided for this question not only explained the correct answer, but considered each distracter and explained why it was incorrect. The complete explanation for this question is given below:

When dealing with arrays, there are a few things to remember. When created, the value used inside the square brackets indicates the length of the array, or how many elements it can contain. The length counts from a starting point of 1. The INDEX however, begins at 0. Meaning that in this case, where we created our integer array with a length of 15, the valid index values are 0-14.

(C) is the correct answer because:
i = array.length - 1, evalutes to 14. The last index of the array. The conditional statement will go down to AND include 0, the first index of the array, but will not pass this point and go out of bounds. i-- means subtract 1 from i every time it goes around, so every number from 14 to 0 will be a value of i during the loops lifespan.

Why are the other's incorrect?

(A) This loop would crash at the end. i = 0, this is fine, it is the first value of the index and is correct. BUT The conditional inside the while loop is: i <= array.length, which means it can be less than OR equal to array.length, which is 15. The last index is 14, thus when it attempted to find index 15 of the array, it would crash with an out of bounds error.

(B) This suffers the exact same problem as A, but has been rendered in 'for' loop format.

(D) The loop shown for D would not crash, but nor would it completely cycle through all values of this array. int i = array.length - 1 as discussed above will result in 14 which is correct for the last index of our array. However, the conditional: i > 0 will not ever allow this loop to check index 0. It will stop after cycling through 1.

(E) This loop again will not crash, but will notcycle completely through all values of this array. int i = 1 means that 0 will not be evaluated. the conditional inside the while loop will stop the cycle correctly at 14 to prevent the crash. i++ means that it will increment the index until the conditional stops this loop.

Poor (but useful) questions

As illustrated again later in Section 4.2, even questions containing errors can end up being useful learning resources thanks to the comments written by other students. We have selected an example of a poor question, but one which has actually benefitted not only other students but the author themselves. Generally, questions with poor ratings do not get a large number of responses. The primary reason for this is that students are able to sort the unanswered questions by rating, and so tend to answer questions with good ratings while avoiding questions with poor ratings. Figure 1, which plots the number of responses a question has received against the rating it has been given by students, shows this trend clearly.

The question below was answered by 14 students and rated by 10 of them, and was given an average rating of 0.4. This was the very worst rated question of all questions with at least 10 ratings.

What is the appropriate boolean variables need to be stored in A and B if the following returns false:

!A || B && !B || A

Although there are some grammatical errors in the question stem, the concept for the question is fine. However, the author had misunderstood the order of precedence of the boolean operators and the suggested alternative was not correct. This was pointed out in several of the comments to the question, two examples of which are shown below:

"Check this page. && is higher than ||.

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/operators.html

so the equation becomes A + !A, which always evaluates to true. So it doesnt matter what values you put into A and B, the expression is never going to be false."

"As explained by the person above me who linked to the sun page, as that expression stands, it cannot be false. In bracket form it would look like: (!A || B) && (A || !B). Of the answers you gave, none of the above is the correct one. :P "

Despite the error in the question, there is evidence that the associated comments and link to Sun's website has transformed it into a useful learning resource. Not only has this question helped other students, but it even corrected the misunderstanding that existed with the author of the question. One student, who answered the question incorrectly, wrote:

Wow that actually helped me alot lol. Totally forgot about the order of && and ||

The question author responded to one of the comments stating:

Sorry everyone..thanx for the reply..i've posted the new version of this question. Feel free to check it out n comment on it (i've 'repaired' my understanding, i hope i got it right this time :D)

Complex questions

From an instructor perspective, good questions are often those which isolate a concept to identify where misconceptions are occurring. Such questions inform the instructor about student performance and thus are useful for summative assessment purposes. From a student perspective, a complex question that involves many concepts can be useful for determining their understanding of a diverse range of topics. This can be valuable for self-assessment, as the student will know where they had problems, and the explanation will help them to understand where they went wrong. We present below one example of such a question:

What is the output of the following code?

```
public class Peerwise {

   public void start() {
         int value = 2;
         int x, y, z;
         String output = "";
         x = 3;
         y = x*2;
         z = value++;
         if (methodOne(x, y)) {
                output += "1";
                z++;
         }
         if (methodTwo(x, y, z)) {
                output += "2";
         } else {
                output += "3";
         }
         while (output.length() > 0) {
                System.out.print(output);
                output = output.substring(1);
         }
   }
```

```
    public boolean methodOne(int y, int x) {
        return y - 3 == x;
    }

    public boolean methodTwo(int x, int y, int z) {
        x++;
        y += y;
        double d = ((y / x) + 1.0) % z;
        return d == 2.0 || d == 0.0;
    }
}
```

The distracters for this question were effective with four out of five of them receiving at least one response. The explanation was also good, and stepped through most of the lines of code. For summative assessment purposes, this question would be of little value to instructors as it would be hard to determine the source of any misunderstanding from an incorrect answer. To a student, answering correctly provides positive feedback about their understanding of a multitude of topics. With the associated explanation, if a student answers incorrectly, they should be able to identify their error. There was one comment written about this question:

That was an awesomely awesome question

It is hard to tell whether this is sarcastic or an exaggeration, but the comment was agreed with by another student and the question received a very good rating (3.75).

Good questions, improved by others

There are several examples in the repository of good questions except for the fact that they lack quality explanations. We note that even questions submitted without a good associated explanation can still be a valuable learning resource thanks to the comments left by other students. The last of our case studies highlights one such example. Consider the following question:

What is the output?

```
Point[] pts = new Point[3];
pts[0] = new Point(100, 300);
pts[1] = new Point(200, 200);
pts[2] = new Point(300, 100);

pts[1] = pts[2];
pts[2] = pts[0];
pts[0] = pts[1];

System.out.println(pts[0].x + " , " + pts[0].y);
System.out.println(pts[1].x + " , " + pts[1].y);
System.out.println(pts[2].x + " , " + pts[2].y);
```

This question was well answered, with 74% of 27 students answering correctly. The explanation given by the author of this question was:

```
pts[1] = pts[2];    pts[1] --> 300 , 100.
pts[2] = pts[0];    pts[2] --> 100 , 300.
pts[0] = pts[1];    pts[0] --> 300 , 100.
```

This does attempt to show the references from the array elements to the Point objects, but it does not discuss the details of reference variable assignment. Another student submitted the following comment as an attempt to clarify this:

The explanation is lacking. It should be made quite explicit that when you make an assignment of one object equalling another object, that the pointer value is copied exactly at that moment in time. It does NOT create a link to whatever the object on the right side of the expression happens to equal after the expression took place. i.e, If we say that:

Pts[0] points to memory address starting at AA0000

Pts[1] points to memory address starting at BBAA00

Pts[2] points to memory address starting at CCBBAA

Then when we perform the expression: Pts[0] = Pts[2]. we are saying copy the memory address in Pts[2] RIGHT NOW. So Pts[0] is now equal to CCBBAA. If we then say Pts[2] = Pts[1], then Pts[2] changes memory address to: BBAA00.

The author of the question posted a reply to this comment, thanking the contributor of the explanation for the improvement:

Thanks for the detailed explaination! Hopefully this helps the other members that are still confused after my pretty vague explaination (sorry! XD)

4.2 Error discovery and correction

As mentioned in Section 4.1, the correctness of the student generated questions is of great importance to instructors. A repository of questions in which a high proportion are not correct would be of little value and it could be argued may even be misleading to students. We randomly selected 61 questions, approximately 10% of the repository, and examined each for correctness. We found 89% (54 out of 61) of these questions to be without error and indicating the correct answer. While this is a positive result, we felt it was important to investigate in more detail the seven questions that were found to be incorrect. In particular, we were interested in the ability of the students using PeerWise to detect, and correct, these errors.

We discovered that in all seven cases the errors were discovered and commented on by students. In addition, where a correct alternative was available, in each case it was the most popular one selected by students.

While our sample of 61 questions represents only 10% of the repository, the results we have seen give us confidence in the students' ability to detect and discuss errors in the small percentage of questions that contain them.

The average rating given to these seven questions by the students was 1.7, considerably lower than the average rating in the repository. The low ratings assigned to these questions reduces their prominence in the repository, as we have previously noted that questions with low ratings are answered less often.

We include here one of the seven questions that contained an error and the comment from a student that identified the error and offered a correct explanation.

This question was on method tracing, and asked for the output of the following code:

```
public void start () {
    int number = 3;
    calculation(number);
    System.out.println ("1: " + number);
}

private void calculation (int number) {
    number = number + 5;
    System.out.println ("2: " + number);
}
```

A set of plausible distracters was offered:

A	B	C	D
1: 8	*2: 8*	*1: 3*	*2: 8*
2: 8	*1: 8*	*2: 8*	*1: 3*

The question author's error was in selecting C rather than D as the answer. The most popular alternative was in fact the correct response D, which was chosen by 42% of the 12 who answered this question. There was also an excellent comment, which was agreed with by another student, pointing out the error and clarifying with an example:

Sorry, but you are incorrect in the order. While your numbers are correct, 2 will print before 1. The calculate method will run in it's entirety (because there are no early return points) BEFORE the rest of the calling point's code is execute. If you were to write that block of code inline, it would look like this:

```
public void start () {
    int number = 3;
    // replacing the calculate method call:
    System.out.println ("2: " + (number + 5));
    System.out.println ("1: " + number);
}
```

4.3 Question styles

The structure of the questions written by students mirrored the questions used by academic staff in their local context (i.e. in the tests and exams of the institution). Furthermore, the subjective quality ratings assigned by the students correlate well with the overall quality rating of the instructors involved in teaching the course.

However, other teaching contexts are different. The Leeds study used questions with a different characteristic structure. The correlation between the staff rating of quality and student rating of quality was lowest when the staff member was not involved in teaching the course. It appears that students adapt to the context in which the course is taught and create questions that have similar structure and focus on similar topics to questions that they have seen during class and in tests and exams.

We expect that if PeerWise was used in a different teaching context, then students in that environment would mirror the structure and focus emphasised by the teaching staff in that local context. To verify this prediction, we could replicate this study at a different institution with different teaching staff. If we don't see similar trends, then it might show that students are having difficultly understanding what the instructor wants them to learn, or alternatively, it might be related to the construction of MCQs themselves (i.e. it might be easier to create MCQs about particular topics, and it might be easier to create questions that have a particular structure such as asking "what is the output of the following code?").

Further investigation of the question style and quality in different teaching contexts is required to better understand how students decide to structure their questions.

5 Conclusions

Students are capable of writing questions that faculty judge to be of high quality. The best questions have well written question stems, good distracters and detailed explanations that discuss possible misconceptions.

Although the questions created by students do vary widely in quality, students are quite capable of making accurate judgements of quality and rate the questions appropriately. Student judgements of quality correlate well with faculty when overall subjective judgement is required. Since students use the rating to determine which questions they answer, we are confident that students view the high quality questions more frequently than the low quality questions.

Inspecting a sample of the questions more closely revealed that most of the questions were clear and unambiguous, free from grammatical or other minor errors, had a high number of feasible distracters and had a good explanation.

The structure of the questions written by students is generally similar to that of the questions written by faculty teaching the course (i.e. for examination purposes in mid-semester tests and final exams), suggesting that students are sensitive to the local teaching context and will create questions appropriate for the style of the questions presented in the course.

The majority of questions have correct solutions (i.e. the answer suggested by the author is the correct answer), and in the cases where the solutions are incorrect, other students are able to identify the errors and offer a correct solution or explanation. This suggests that most learning experiences with PeerWise should be error free for students.

6 Future work

We plan to further investigate the issue of question quality by replicating the study at a different institution where the instructor has a different teaching style and focus.

No instruction has been given to students with respect to the creation of MCQs. It would be interesting to investigate whether formal instruction about creating questions, choosing appropriate distracters and the importance of an explanation has an affect on the quality and structure of the questions. Altering the assessment criteria for the contributed questions to explicitly require an explanation might reduce the number of questions that lacked explanations entirely and may generally improve the quality of the explanations produced.

7 References

[1] P. Denny, A. Luxton-Reilly, and J. Hamer. The PeerWise system of student contributed assessment questions. In Simon and M. Hamilton, editors, Tenth Australasian Computing Education Conference (ACE 2008), volume 78 of CRPIT, pages 69-74,Wollongong, NSW, Australia, 2008. ACS.

[2] P. Denny, A. Luxton-Reilly and J. Hamer. Student use of the PeerWise system. In ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education, pages 73-77, Madrid, Spain, July 2008. ACM.

[3] P. Denny, J. Hamer, A. Luxton-Reilly, and H. Purchase. Peerwise: students sharing their multiple choice questions. In ICER'08: Proceedings of the 2008 international workshop on Computing education research, pages 51-58, Sydney, Australia, 2008.

[4] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Mostrom, K. Sanders, O. Seppala, B. Simon and L. Thomas. A multi-national study of reading and tracing skills in novice programmers. SIGCSE Bulletin, Volume 36, Issue 4 (December 2004), pp. 119 - 150. ACM.

[5] J. Sitthiworachart and M. Joy. Computer support of effective peer assessment in an undergraduate programming class. Journal of Computer Assisted Learning (2008), 24, 217–231

[6] K. Topping. Peer assessment between students in colleges and universities. Review of Education Research (1998), 68 (3), 249-276

Easing the Transition: A Collaborative Learning Approach

Katrina Falkner **David S. Munro**

School of Computer Science
University of Adelaide,
Adelaide, South Australia 5005,
Email: {katrina.falkner,david.munro}@adelaide.edu.au

Abstract

Engaging first year students is a difficult problem, as students must develop independent study skills while concurrently mastering their chosen topic. At the same time, they find themselves in an alien environment, removed from their peer group and anonymised by University structures. Retention is of particular concern within ICT as across Australia, and globally, we have seen a recent dramatic drop in number of students applying for ICT degrees, and poor progression and retention rates.

Collaborative learning is a strategy that involves the students themselves in the ownership and direction of their learning; each student is responsible for not only their own learning but of the learning of the group. In this paper we describe our approach to student engagement based on applying a range of collaborative learning techniques within an introductory Computer Science course, addressing specifically the task of collaborative problem solving. Results from three years of adopting this change in teaching methodology indicate increased student confidence, participation and student ability.

Keywords: Computer Science Education, CS1, Transition, Collaborative Learning.

1 Introduction

Engaging first year students is a difficult and well-recognised problem, as students must combine the development of independent study skills with mastery of their chosen topic. Although already a difficult prospect, this is further complicated by the transition to University education, requiring most students to enter an alien education system, consisting of large classes, a de-personalised administrative system and separation from their peer group. Studies indicate that students feel "part of an anonymous mass" (White 2006) and indicate that one of the most common reasons for leaving the University education system is a feeling of isolation and loneliness (Burns 1991). Our classrooms are increasing in diversity (Biggs & Tang 2007), further complicating this issue by presenting us with an "unprecedentedly broad spectrum of student ability and background" (Ramsden 2003). In addition, students have increased expectations on their ability to co-ordinate study, part-time work and external activities (Krause et al. 2005), requiring mature time management and study skills. Forming of bonds within the peer group at the commencement of University education addresses a common cause of transition distress. 30% of students who continue with their studies report that the establishment of a peer group was the significant contributing factor (Cameron 2007).

We have observed three interrelated issues that impact upon a student's initial success at University: engagement, transition concerns and attendance. These issues are strongly connected: students suffering from transition are likely to miss classes and be distracted in those that they do attend. Missing classes impacts upon engagement directly, as students who fall behind can easily become disillusioned about their ability to succeed at University. Conversely, students who participate and engage in the classroom are more likely to form bonds with their peers, overcoming transition issues and encouraging them to continue with their studies (McInnis et al. 2000). Increasing student engagement and attendance has the benefit of decreasing attrition, but has a perhaps more significant impact on student performance. Studies have shown a strong correlation between attendance and student performance, both within traditional didactic lecture style courses, and courses based around active, or collaborative learning principles (Urban-Lurain & Weinshank 2000).

Engagement of first year students is of particular concern within the ICT discipline - within Australia, and globally, we have seen a recent dramatic drop in applications for ICT degree program, accompanied by poor progression and retention rates (Sheard et al. 2008). The understanding of programming poses many challenges for first year students, as a large number enter ICT programs with little or no prior experience. Novice computer science students are required at an early stage to develop a diverse range of skills: problem analysis, problem solving, code development and testing, and then integrate those skills within the software development process. Studies show our students continue to struggle with this task, even though it is common for Universities to dedicate a disproportionate amount of resources towards their learning (McCracken et al. 2001, Lister et al. 2004). Lack of performance in programming has been seen to impact other facets of their study, including their overall program progression, confidence and study habits, all factors that affect engagement and retention.

A further issue is the decreasing percentage of female students seeking to study ICT. Despite increasing resources dedicated to this problem, numbers continue to decline, leading to low, and decreasing, numbers of women seeking employment in the ICT industry. During their studies, lack of confidence has been identified as a significant challenge for female students, leading to deferral to their male peers and a reluctance to experiment (Trajovski 2006). Differ-

Copyright ©2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 95, Margaret Hamilton and Tony Clear, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

ences can also be seen in technical success - a recent study reports that the highest failure rates for female students within ICT are in technical courses (Lang et al. 2007), although overall study completion times are similar (Ilian & Kordaki 2006).

In this paper we describe the use of *Collaborative Learning* principles (Smith & MacGregor 1992) in the teaching of an introductory Computer Science course. Collaborative learning defines a range of educational approaches that involve group work, with students being responsible for not only their own learning but also that of the group. In these approaches, the emphasis is constructivist, and based on the student's experiences in application of course content rather than in their observation of course content. Collaborative learning is well established as having significant social benefit beyond that of academic development, with involvement in learning relationships with other students and academics having a significant impact upon student engagement and attrition (MacGregor 1991, McKinney & Denton 2006).

We explore collaborative learning techniques in the following ways:

- Structured collaboration initiated within lecture and tutorial classes based around the task of problem solving in programming.
- Peer-based support structures designed to extend collaborative learning approaches outside of the classroom, and foster the formation of *learning communities.*

Many students - and some academics - are believers in the idea of the *super programmer*: that expertise at programming is an inherent skill that can not be learnt. A lack of success in their early attempts can thus be debilitating for first year ICT students. However, to the contrary, research indicates that expertise corresponds to the repeated application of theory and repeated practice, even among elite performers (Ericsson et al. 1993). Further, cooperation with colleagues - working as a group rather than alone - is evidenced as key requirement for expertise in software development (Sonnentag 1998). Working collaboratively is an integral part of being an ICT professional.

We employ the *action research* methodology (Hopkins 1985, Carr & Kemmis 1986) in the construction of this project: an iterative research process that involves research with - rather than on - the participants. This paper describes our results over two iterations of the research process, encompassing three years of experimentation with collaborative learning techniques.

2 Collaborative Learning

Collaborative Learning encompasses a range of constructivist learning techniques where students work in groups, sharing and constructing their knowledge within a common aim. Collaborative learning is known to be effective in academic development, as students are more engaged with the subject matter, but also supports community and social development. It is seen as a particularly relevant technique in first year education, in that it addresses transition concerns by increasing deeper learning and a sense of belonging (McKinney & Denton 2006).

Collaborative inquiry is supportive of learning for all parties involved, not simply through the sharing of knowledge but in the elaboration and discussion that ensues. Glasser draws a distinction between learning by observing and learning by doing and being involved:

"Students learn 10% of what they read,
20% of what they hear,
30% of what they see,
50% of what they see and hear,
70% of what is discussed with others,
80% of what they experience personally, and
95% of what they teach to someone else"

(Glasser 1990)

Examples of collaborative learning include *cooperative learning* (Slavin 1983, 1991), *problem-based learning* (Barrows 1986), *discussion groups* (Christensen et al. 1991) and *peer teaching* (Jenkins & Jenkins 1987). Cooperative learning is a heavily structured form of collaborative learning, where students participate in groups with a significant emphasis on interpersonal skills and assessment of management roles. This approach adopts cooperative incentive structures, where the success for the individual depends on the success of the group as a whole. Problem-based learning involves the immersion of students in complex, real-world problems where they must work together to both identify what they need to learn in order to solve the problem, and undertake this learning collaboratively. Again, this approach emphasises teamwork and problem solving skills, and may involve role play or simulations.

Discussion groups can be either formal or informal and involve a discussion led by both the teacher and the students around a specific topic. The aim here is to open dialogue and explore an open-ended discussion, enabling each student to express their views and for students to debate and learn from each other's opinions. Peer Teaching, perhaps better named *cross-age teaching*, involves more advanced students acting as experts employed to instruct and mentor novice students. Although not strictly *peers*, the two students are generally close in age and share in the same experiences. There is no boundary of authority between the two, with no threat of assessment, leaving the novice able to openly express their concerns. Mentoring programmes, particularly in academic settings, act as informal peer teaching approaches.

The broadest application of collaborative learning is the development of learning communities, where curriculum are restructured to form collaborative student groups that cross course boundaries, leading towards the development of a *Community of Practice* (Wenger 1998). Learning communities build coherence within individual programmes, and foster informal generation of peer teaching programs.

A successful collaborative learning structure does not impose structures based on ability or previous experience, but instead supports individuality and diversity. Academics work with students on setting goals and tasks, with the outcomes of individual groups used in the coordination of topics and discussion. An essential aspect of a collaboration is the authenticity of the task that they are attempting to solve, and the process by which they solve it. Mutual engagement requires that each member have a genuine motivation for being part of the collaboration, and that learning be associated with a task or problem that the group has to solve. We have selected problem solving as the task around which we construct our collaborations.

What does it mean to teach an *authentic* problem solving process? The development of software is a creative process, requiring design, innovation and communication skills. Software developers commonly start with a partially defined problem, and must communicate with clients and colleagues in determining the full problem specification. This is just the beginning, however, as what follows is an extensive process of creation, partial solutions and experimentation to determine the final solution. Introductory classes in Computer Science, however, tend to take the opposite approach; we focus on learning syntax and semantics, and the problem solving process as one of immediate success. Very rarely do introductory Computer Science texts elaborate examples that illustrate that solving problems most commonly starts with failure. This is detrimental for our students, as studies indicate that students tend not to vary their approach to problem solving, instead persisting in applying the same techniques regardless of their success (Ginat 2001).

Further, our approaches to teaching introductory programming commonly place an emphasis on learning the application of single programming concepts, using known-answer assignments as demonstration. As such, these kinds of problems are valuable, as they enable students to observe and practice the application of individual concepts. However, this ignores the skill of the problem solving process itself, and does not provide students with a realistic illustration of software development. Indeed, this approach constrains us further in that it does not provide opportunities for exploring variations of problems and how the problem solving process might apply in these new situations.

As software developers we do not solve problems on our own, or attempt to solve problems that we already know there is an answer to. However, we expect our students to learn how to develop software by doing just that. If we hide the problem solving process away as something we do between lectures, then students stand little chance of learning it from us. Our approach consists of exposing, and forming learning communities, around the problem solving process.

3 Establishing Computer Science Learning Communities

A comprehensive programme of collaborative learning techniques has been introduced in an introductory computer science class in order to address the issues of student engagement and transition concerns. This changes the style of education from the traditional didactic lecture style to a more interactive, student-driven approach.

We employ a range of collaborative learning techniques within the course activities and associated support systems. The course of approximately 200-250 students contains a mixture of domestic and international students (approximately 50% of each), and is taken by students from a broad range of disciplines, including Computer Science, Engineering, Commerce and Arts. The majority of students who take this course are entering University straight from the secondary education system and are new to the process of adult, or self-directed, learning. Students are taught the fundamentals of programming and an introduction to the object-oriented paradigm using the Java programming language.

We employ the *action research* methodology in our approach - this paper describes two iterations of the research process over a duration of three years.

3.1 Research Methodology

Action Research is an iterative research methodology that involves the subjects of the research as active participants - the research is done with, rather than on, the participants (Hopkins 1985, Carr & Kemmis 1986). According to Carr and Kemmis (Carr & Kemmis 1986), action researchers "see the development of theory or understanding as a by-product of the improvement of real situations, rather than application as a by-product of advances in 'pure' theory'. In other words, action research is research directed towards the improvement of practice.

Action research involves repeated application of stages of planning, action and observation, and reflection (see Figure 1), with the perceptions of the participants included in the reflective exercise. It is an appropriate research methodology for use when a problem has been identified, an initial plan has been constructed and further strategies that may be employed depend upon the success, or impact, of the initial strategy.

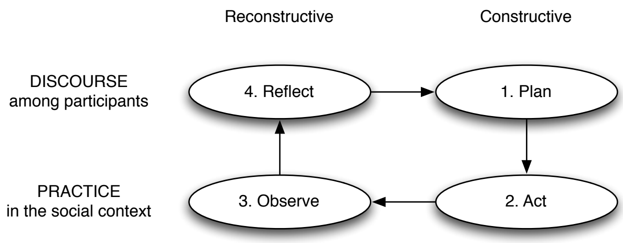


Figure 1: Action research spiral (Carr & Kemmis 1986)

An important aspect of action research is that the participants are informed of the research, its motivations and rationale. The participants are then able to reflect, along with the researcher, on the success of the research project. In order to engage the students in the learning approach, the course starts with a discussion of learning styles, and the benefits of collaborative learning, discussing the work of Glasser (1990) and Slavin (1983).

3.2 Iteration 1: Authentic Learning Practices

Our initial plan was based around the introduction of a range of collaboration learning activities, utilising a variety of collaborative learning techniques constructed around the task of problem solving. In an introductory Computer Science class we wish students to learn the different programming concepts, how to apply each and how to compose them to solve a larger problem. Problem solving skills are used both in the identification of the specific programming concepts to use in solving a problem, and in directing how programming concepts should be composed.

Our initial application occurred in 2006, subsequent to having been involved in the instruction of this course since 2004, and involved the following activities:

- Discussion-driven Problem Solving Lectures
- Collaborative Tutorials
- Cross-Age Teaching Support Systems

Lecture classes are an hour in duration, held three times a week and include all students in the class. In addition, students attend weekly practical sessions that are two hours in duration, and weekly tutorial classes (one hour in duration). Lecture classes adopt a combination of observation using known-answer questions, and experiential learning. These are designed to encompass the spectrum of the problem solving process: observation of application of programming concepts and authentic problem solving processes, and discussion driven problem solving activities. Problems are chosen representing the spectrum from known-answer to open-ended, enabling practice as well as exploration of the problem space. A fundamental element of all lectures is the use of computer-assistance, demonstrating the application of concepts and the development of solutions using the same laboratory environment and tools that the students will use themselves.

We demonstrate the problem solving process through the integration of *Problem Solving* and *Worked Examples* lectures at key stages where students must learn to compose programming concepts, and to solve more abstract problems. These lecture models provide two different styles of discussion driven collaboration; problem solving lectures emphasise demonstration and provide more scaffolding of the problem solving process, while worked examples lectures are more under the control of the students, enabling them to lead the discussion.

In Problem Solving lectures, the stages of the problem solving process are demonstrated: elaboration of the problem, identification of key elements in the solution, and identification of key classes and their responsibilities and necessary algorithms. The lecturer leads whole-class discussion in each stage, with individual students contributing questions and design ideas at requested points. An important aspect of these lectures is to explore the potential problem space, rather than focusing on one planned pathway. Multiple solutions, based on the inputs of the students are presented; two or more potential solutions are then taken from the design phase through to implementation, enabling discussion of the negatives and benefits of each selected solution.

The process in these lectures is to start with a high level, and incomplete, problem specification, for example:

> We have been contracted to develop a new mobile phone. We decide to build a software simulation in Java of how it is going to work.

A common design aspect targetted by the students is the design of an *Address Book*. Figure 2 illustrates two common designs identified by students. Figure 2(a) illustrates a simple design, where an address book consists of a set of (name, phone number) pairs; Figure 2(b) presents a slightly more complicated model, where an address book consists of a set of entries that model a person, while a person may have a set of phone numbers. The second design is more elaborate, but has the advantage that multiple phone numbers that belong to the same individual are easily identified, making subsequent computation simpler.

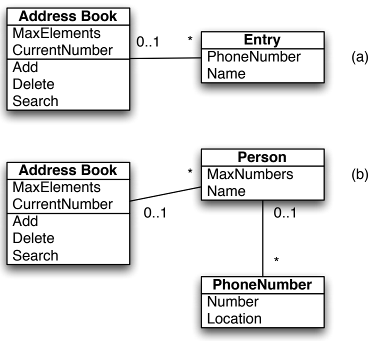


Figure 2: Alternative Designs

In Worked Examples lectures, students are presented with a small set of problems that they must solve as a group. Adopting similar processes to the problem solving lectures, the students work through the different stages of understanding and solving the problems (typically simpler than those used in the problem solving lectures). A key distinction between these lecture styles is that within worked example lectures, the students drive the process with the lecturer acting as scribe.

Not only correct solutions are educational; exploring poor design choices, such as inappropriate data type selection or class design, enables students to observe the impact of such choices. This not only assists them in identifying their own poor design choices, but enables them to observe failure and recovery as a normal part of the software development process. Exploring wrong directions and the results of failure can be important in establishing a starting point for divergent thinking, which stimulates creativity (Ginat 2008).

To extend our example, students commonly discovered problems with their design when discussing appropriate data representation for a phone number. Many students initially suggest an integer representation; once they work through an example illustrating the representation of the number, however, they soon realise that that is inappropriate.

In addition to discussion driven collaborative lectures, we also introduced collaborative tutorial sessions, and collaborative support services. Weekly tutorials are held in a similar style to the discussion driven lectures. Students are expected to submit answers to a set of tutorial questions prior to the tutorial, which are then assessed by the tutor. Within the tutorial session, the students work in small groups to elaborate on their answers with each group working on a different problem. Each group then presents its solution to the rest of the tutorial class. We introduced a training program for all casual academic staff involved in first year tutorials and practical sessions in 2006, providing an introduction to collaborative learning and small group teaching techniques. The structure of the training programme adopts collaborative learning techniques itself, in that participants learn and reflect on their teaching skills through the use of small group role play sessions. All tutors and practical supervisors must attend this program, regardless of previous attendance. This provides further opportunity for reflection for the experienced participants, and the opportunity for cross-age teaching for the novices.

It can be difficult to provide appropriate support services for first year University students, as many are shy and lacking in confidence. Beder reports that first year students are simply not aware of support services on offer, and do not take advantage of them (Beder 1997). We introduced the *Computer Science Learning Centre* as a dedicated space for cross-age teaching. The learning centre is a space positioned within the first year laboratory environment where novices may go for help on any aspect of studying Computer Science. The Learning Centre is staffed by senior undergraduate students who have undergone the same small group and cooperative learning training as our tutors. The Learning Centre staff act as informal tutors, providing help and advice on topics ranging from Java fundamentals to exam study techniques. The Computer Science Learning Centre is now the focal point for students within the School, with senior students often volunteering their time, beyond their scheduled hours.

Webb (1989) identifies six conditions that are needed for successful, and thus educational, cross-age teaching: (1) The tutor must provide relevant help which is (2) appropriately elaborated, (3) timely, and (4) understandable to the target student; (5) the tutor must provide an opportunity for the tutee to use the new information; and (6) the tutee must take advantage of that opportunity. As we are focusing on the very practical discipline of problem solving and programming, we have constructed the learning centre to reflect the laboratory environment of the student's normal practical sessions. The space is divided into two sections: one section containing computers that provide an identical laboratory environment, and the other a general study and discussion area. This model enables novices to move between teaching and application phases without leaving the space, and the care of the expert.

Student perceptions of ability, success and understanding are indicators of their confidence and engagement. Qualitative analysis of student perceptions of the introductory computer science course using student experience surveys indicate little or no change in student opinion on issues such as motivation to learn, feeling part of a group and enthusiasm for learning. Table 1 illustrates the results of surveys taken from 2005 to 2006 showing broad agreement with a series of 7 point Likert style questions over this period, where broad agreement is the percentage of students who have responded positively (not neutrally) to the question. Although students indicated in these surveys that they were actually *less* enthusiastic about their learning, a significant number made positive comments regarding the new collaborative learning activities. 25% of the comments describing the best aspects of the course related to its approach to problem solving, for example:

> The worked examples are a good idea. Get to see the concepts used in a practical sense.
>
> The demos. Give us something to look at and see what the program does.
>
> Problem solving to complete what is required in the pracs and seeing a working program.

Question	2005	2006
Q1: I feel part of a group committed to learning	39%	43%
Q2: This course stimulates my enthusiasm for further learning	68%	57%
Q3: I am motivated to learn in this course	59%	60%
Q4: This course helps me develop my thinking skills (eg. problem solving, analysis)	77%	79%
Q5: I understand the concepts presented in this course	61%	64%

Table 1: Student survey results from 2005 and 2006.

Figure 3 illustrates the practical examination performance from 2004 to 2006 (the time of the first iteration). The first practical examination assesses fundamental computer awareness skills and the ability to use the practical laboratory. Questions in this examination are relatively simple: requiring correction of some simple syntax errors, and demonstration of the ability to edit and compile source code, and the use of various tools required to execute the compiled code. Failure in this examination is used as an indicator of a lack of confidence in using the laboratory facilities, and of requiring assistance in understanding instructions, and does not address problem solving. We see the improvement in average performance in the first practical examination as a result of incorporating live demonstration, including experimentation and examples of failure.

Figure 3 also demonstrates results for the second practical examination (PE2), which assesses problem

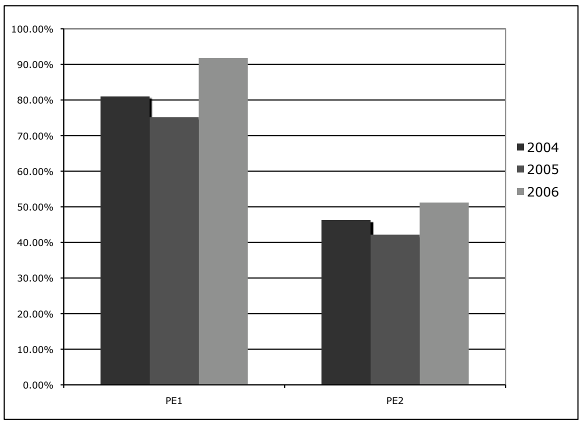


Figure 3: Practical Examination performance 2004-2006 (after first iteration).

solving skills and the application of programming concepts. At this stage, no significant improvement in problem solving skill was indicated by the practical examination results.

Figure 4 illustrates attendance rates for practical classes and lecture classes, and attrition rates for 2004-2006[1]. As can be seen, no significant impact can be seen across the course offerings.

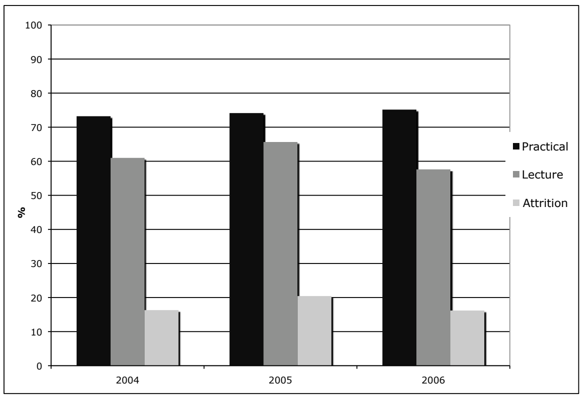


Figure 4: Attendance and attrition rates from 2004-2006 (after first iteration).

In our initial application of discussion driven problem solving, we adopted a co-lecturing approach with two lecturers engaging the class in discussion to explore the problem space. Students indicated in their reflections that they enjoyed these lectures, and found the collaborative problem solving useful in demonstrating real-world problem solving processes. However, we felt that the students were still acting as observers rather than collaborative participants and did not naturally take a leadership role in driving the problem solving. We observed that there was no increased incentive for them to attend, and engage in, lectures as they were not truly involved in the problem solving processes and had no interdependence on their peers. Further, we had created stronger peer relationships outside of lectures, through our modifications to tutorials classes and the introduction of the learning centre.

3.3 Iteration 2: Cooperative Learning Practices

In our second iteration (2007-2008), we retained the collaborative learning activities provided by tutorials

[1]As tutorial attendance is a compulsory component of course assessment it is not relevant as an indicator of student engagement.

and the learning centre, and refined those offered in lecture classes to move towards a cooperative model rather than a discussion driven model. In cooperative learning, there is incentive on students working together - this incentive can be formal, and can include an assessment component, or it can be informal, and driven by a sense of social responsibility. We have adopted the latter approach.

From the first lecture class, we reorganised students into small groups of 3 or 4 students. Worked Examples lectures then involved allocating 2 or 3 problems across the individual groups. Groups were then asked to discuss the problem to make sure that each member of the group understands what was being asked, to design a solution and to start to develop a software implementation of their design. The role of group scribe is rotated within the group throughout the course, giving every student a chance - and the responsibility - to lead the process.

An example problem that we employ here is the design and implementation of a testing interface for the phone simulation introduced in the previous section. Students must discuss what features require testing, and how they will test them. They must also design an interface that will support their tests. Working in groups enables students to share their ideas and their assumptions on how their software should be tested, and enables them to explore this phase of the software development process in a way which is not associated with the pressure of obtaining a working solution for assessment.

In these classes, the lecturer spends their time moving between groups, advising them and answering their questions. This stage enables the lecturer to observe common problems, and to gather examples of different solutions. The last 10 minutes of the lecture is then led by the lecturer, summarising any common issues and presenting one or two example designs. Multiple solutions to the problem were posted to the electronic bulletin board following each lecture to enable the class to continue its discussion.

This stage enables the lecturer to make personal contact with each student (over the semester), and helps identify students who need additional assistance. We were able to identify individual students who required the help provided by the Learning Centre or individual consulting provided by the lecturers outside of class. Private, and informal, discussions could be held between individual groups and the lecturer. In our discussion, students reported that they felt more comfortable in asking questions and arguing their positions in their groups than they felt in larger classes with a tutor or lecturer. They also indicated that they obtained a clearer picture of where they stood relative to their peers, providing sufficient incentive for some students to seek further help outside of class in order to catch up.

Interestingly we found that students remained in their initial groups for the remainder of the semester, regardless of whether they were undertaking group or independent work. We feel this is of wider benefit in an introductory class; forming of bonds within the peer group at the commencement of University education addresses a common cause of transition distress (Krause 2005, Cameron 2007).

Student perceptions on their ability and engagement with the course altered significantly subsequent to the adoption of cooperative learning activities within lecture sessions. Table 2 shows the result of the same student survey referred to in Table 1 for 2008 showing broad agreement with a series of 7 point Likert style questions, where broad agreement is the percentage of students who have responded positively (not neutrally) to the question. (Data is shown for 2005, for comparison, with 2008 data).

Question	2005	2008
Q1: I feel part of a group committed to learning	39%	58%
Q2: This course stimulates my enthusiasm for further learning	68%	74%
Q3: I am motivated to learn in this course	59%	71%
Q4: This course helps me develop my thinking skills (eg. problem solving, analysis)	77%	88%
Q5: I understand the concepts presented in this course	61%	80%

Table 2: Student survey results from 2005 and 2008.

Student comments indicate that they appreciate the opportunity to work together within lectures, with the accompanying realisation of the application of programming theory, identifying the best aspects of the course as:

> Worked examples because they are good, especially in small groups.
>
> Inclusion of students into lectures.
>
> Worked examples lectures are extremely helpful in allowing students to see the theory being put into practice.

Figure 5 illustrates the change in broad agreement, along with the provisional variation from aggregated data for all courses within the University for 2008[2]. This data demonstrates the improvement resulting from the adoption of cooperative learning techniques, and demonstrates that student satisfaction is within range, or more typically, higher than University aggregate data.

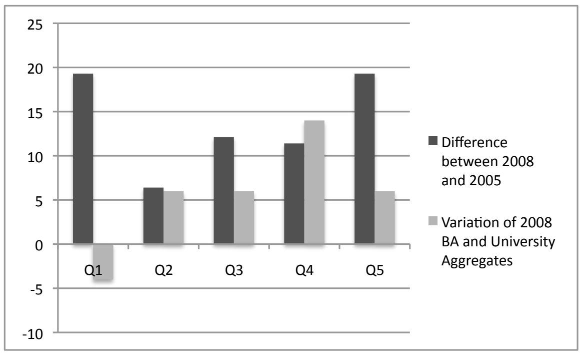


Figure 5: Comparison of survey results from 2005 and 2008, and variation of 2008 survey results from University aggregate data.

Figure 6 shows practical examination performance from 2004-2008. Unsurprisingly, performance in the first practical examination does not change significantly, as students are still engaged in the same authentic problem solving and live demonstration processes. However, after initiating cooperative learning techniques within lecture, performance in the second practical examination, which requires greater problem solving skill, is significantly improved. Figure 6 illustrates continued improvement across the two years of the second iteration of our study. In 2008, our second application of cooperative learning, we provided more structure in terms of identifying roles within the groups and also demonstrated the process of problem

[2]Aggregated data from 2007 was used for provisional comparison.

solving cooperatively by combining discussion-driven segments within the worked examples lectures. This scaffolding provided additional incentive and knowledge for student groups that were struggling with either group dynamics or the problem solving process.

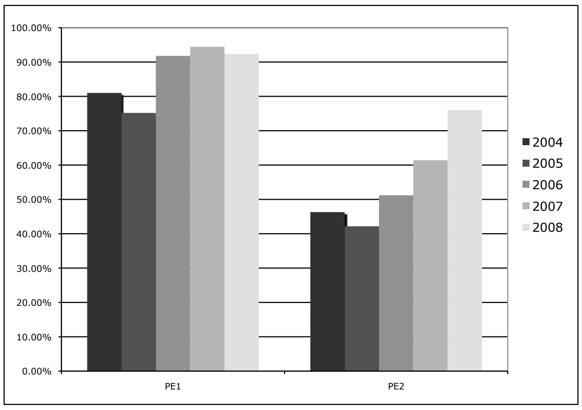


Figure 6: Practical Examination performance 2004-2008 (after second iteration).

Figure 7 illustrates attendance rates at practical classes and lectures from 2004-2007[3], combined with course attrition rates. Minor, but steady, increases can be seen in attendance at practical classes, but a significant increase in lecture attendance can be observed in 2007 upon the introduction of cooperative learning techniques.

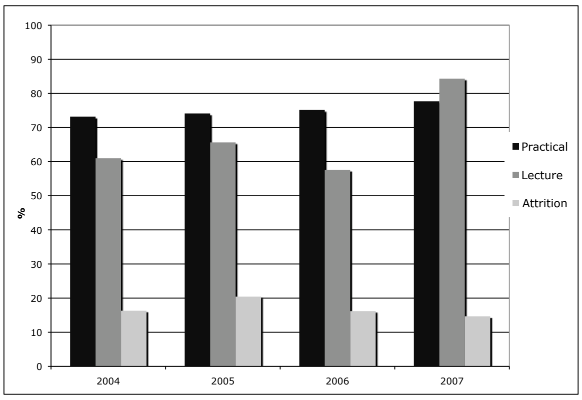


Figure 7: Attendance and attrition rates from 2004-2007 (after second iteration).

Out-of-class consulting is provided in each offering of the course, through in-office consulting hours for lecturers, and an on-line discussion forum. Since the introduction of the cooperative learning activities, students have engaged more with the course both within, and outside of, class. The number of students active on the discussion forums has increased from 44% to 93%, and the number of accesses averaged per student has increased from 4.9 to 191. Students have adopted the principles of cooperative and collaborative learning to create their own learning communities outside of the classroom, including establishing online problem solving forums where students set their own problems and rules for success, such as fastest execution time or earliest correct submission. Interestingly, both cohorts of students independently developed a variation on these forums in response to exposure to cooperative learning approaches.

[3]Attendance figures for 2008 are unavailable.

3.4 Summary

This paper presents two action research cycles designed to address student engagement through the adoption of collaborative learning approaches, constructed around the topic of problem solving. Both action research cycles adopted multiple collaborative learning strategies, designed to provide multiple opportunities for student engagement, and recognising that a single approach to engagement is unlikely to be successful with current diverse intakes into first year University courses. Two collaborative activities, collaborative tutorials and cross-age teaching support systems, were retained across both action cycles, with the in-lecture collaborative activities forming the focus of change. Table 3 presents a summary of the in-lecture collaborative activities, and their relative successes, employed in each action cycle.

	Cycle 1	Cycle 2
Problem Solving	Discussion-driven with lecturers driving the discussion and demonstration.	Discussion-driven with both lecturers and students driving the development process.
Worked Examples	Discussion-driven with the students together driving the development process.	Cooperative with students working in small groups, followed by open class discussion.

Table 3: In-lecture collaborative activities across both action research cycles.

The significant change from the first to second cycle is a change in teaching philosophy - students may be engaged with the lecturer when the lecturer is enthused and entertaining, however they will be engaged with the subject matter when they are themselves involved in the learning process. This is not a new observation - indeed, it is the basis underpinning collaborative learning - however, it does require rethinking the way we teach, and the way that we structure lectures.

3.5 Discussion

Our introductory computer science class typically contains a broad mixture of students in terms of their previous programming experience. Surveys taken in 2007 indicated that 53% of students had prior programming experience with a high level language, while 55% had studied Information Technology at secondary school. 35% had no prior exposure at all. Since commencing instruction of this course in 2004, we have noticed a general trend of poor attendance by the stronger students, particularly those that have previous programming experience. After the introduction of cooperative learning activities, we noticed that these students continued to attend lectures, even through they also continued to report a degree of boredom with the course content. Students indicated that they enjoyed working in groups, and found the lectures more entertaining even if the material was not challenging for them. We have also found, both in lectures and in the Learning Centre, that the stronger students enjoyed the process of peer and cross-age teaching.

Students that have been involved in these collaborative learning activities have subsequently introduced their own learning communities that extend beyond the introductory computer science course. In 2008, students introduced a computer science student club, whose primary activities are associated with providing cross-age teaching opportunities for other students across all of our programmes and year levels. Since its creation, the club has run a series of orientation events, examination preparation sessions for new students, and a series of technical training seminars associated with learning software tools that are useful in student projects.

Employing a range of collaborative learning techniques has not been without its costs, however. Establishment of the Computer Science Learning Centre was assisted through a University Learning and Teaching Grant, and in-kind contributions from hardware suppliers. This enabled us to run a year-long pilot study creating a dedicated space for cross-age teaching. The supportive reaction of students, both experts (tutors) and novices, involved in the pilot study led us to adopt the Learning Centre as part of our core practice in first year teaching.

Restructuring the curriculum around collaborative learning places an emphasis on regular, sustained contact within the collaborative groups. One lecture every two weeks, at a minimum, is set aside for collaborative work. However, we found that this did not require us to remove any course content. Instead, we were able to explore topics such as problem specification, design and problem solving using collaborative approaches, rather than as didactic lectures. In order to provide more opportunities for sustained collaboration, we doubled the number of tutorials to one per week for the course duration. This enabled us to have multiple tutorials dedicated to significant conceptual issues, such as object-oriented programming, providing time to discuss not only the conceptual issues but also the problem solving concerns.

Each semester we undertake training of all practical supervisors, and tutors in collaborative learning models, requiring time from all staff participating in the instruction of the course. Extensive preparation of a wide range of examples is also required - in addition to the setting of assessment for each course offering, new examples must be introduced in order to keep both the course content and the presentation fresh.

Supporting learning communities introduced an unexpected cost, however, as by including students more openly in the learning process, they then demand more inclusion in other activities. The expanded usage of the discussion forums, and the introduction of the student's own problem solving forums, required significantly more moderation time. Working with the student club, in providing facilities for club activities and staff time associated with assisting with training sessions organised by the club also requires additional staff resources. However, this demand is an indication that students understand and appreciate the teaching methodologies introduced. Supporting these activities is an acknowledgement from the staff that participation from the students in cross-age teaching is valid and appreciated as a general concept, not just within a course.

Adopting such a significant change in our approach to teaching introductory computer science required dedicated time and resources allocated to exploring the pedagogical basis for collaborative learning. Time required to prepare to teach such a course requires not only the development of technical resources, lectures and examples, but also preparation in appropriate techniques for the chosen teaching style. This required the commitment and engagement of all staff involved in the course, from the Head of School (in terms of allocation of free time to develop and research appropriate techniques) and all academic staff involved in the delivery of the course.

4 Related Work

Learning communities and collaborative inquiry have been explored in many different education areas, both in terms of professional development of academics, and within the classroom. Many studies exist within the different collaborative learning styles; a survey of the different styles, and their main examples is provided by Smith & MacGregor (1992). Guidelines for collaborative learning activities within software development are provided by Yerion & Rinehart (1995).

Teague and Roe describe a study of the potential positive effect on learning outcomes by adopting a collaborative learning approach to teaching programming (Teague & Roe 2008). This study is an analysis of student learning preferences and concerns expressed by students as they learn to programme. Although at an early stage and thus without a discussion of its implementation, this study highlights students perceptions that collaborative learning approaches are beneficial and engaging.

In addition, Teague and Roe (Teague & Roe 2008) provide a recent overview of collaborative learning, focusing on cooperative learning, within the ICT discipline. Of note are the results of McKinney & Denton (2006) and (Williams & Kessler 2000). McKinney & Denton (2006) describe a laboratory-based application of cooperative learning, involving students in team-based problem solving, project planning and pair programming, and included early instruction of team skills. The authors find a significant increase in team skills, and in the engagement of their students. (Williams & Kessler 2000) describe a similar attempt at laboratory-based pair programming, with the interesting result that teacher workload was reduced as students were able to rely on their pair colleague for technical support and advise.

Allan & Kolesar (1997) describe their experimentation with a preparatory course in problem solving skills taken before the first programming course. They emphasise that for students without prior programming experience, and from non-computer science majors, a first course in programming can be daunting with the majority of students focusing on learning syntax and "getting the program to run" (Allan & Kolesar 1997) rather than on the more abstract task of problem solving. Their results support our move towards a problem solving approach to programming, although we integrate that with the teaching of programming concepts.

Peer teaching programmes are numerous in the University sector, with examples seen in pair programming in the ICT discipline (as discussed earlier), and cross-age teaching, although little research has been done in the area of cross-age teaching within ICT. D'Souza et al. (2008) discuss a cross-age mentoring programme, where mentors volunteer their time to assist novice programmers in developing their skill. Similar studies have been undertaken by (Miller & Kay 2002) and (Miliszewska & Tan 2007). Although mentoring is provided as a broad experience, with no specific requirement to focus on programming skills, these studies have found that assistance with programming and problem solving are the key issues concerning novice students involved in the mentoring relationships. Peer teaching and mentoring both assist in the creation of learning communities, as they cross course boundaries and form bonds between students that persist outside of an individual course offering.

Strazdins reports on the use of the community of practice model within postgraduate project courses, designed to build research skills in small communities of approximately 10 postgraduate students undertaking research courses (Strazdins 2008). In this study, a community was constructed around the development of general research skills, and resulted in significant improvement of the research experience of those students involved in the community, and improvement in research skill.

5 Conclusions

In this paper we have described the application of collaborative learning techniques within an introductory computer science course. Our aim was to increase student engagement and student ability in problem solving through integrating a range of collaborative learning activities throughout the course structure, including lectures, tutorials and support services. Collaborative learning activities have been recognised as increasing both technical ability, but also as providing opportunities for the development of social support structures. Identifying a peer group and maintaining confidence are well established indicators of ongoing engagement and reduced attrition.

Our first attempt at this approach consisted of a combination of discussion driven lectures, based around demonstrating and participating in the problem solving process, collaborative tutorials and cross-age teaching. Demonstration of the problem solving process, with its emphasis on authentic demonstration using an equivalent laboratory environment, proved successful in engaging students in exploring the laboratory environment and increasing their confidence at early stages in the course. However, our discussion driven lectures appeared to provide no further increase in student engagement or ability, although the students did report an appreciation for these examples.

Our second attempt at integrating collaborative learning activities replaced the discussion driven lectures with cooperative problem solving processes, involving the students in small groups working through a set of small problems. This activity appeared to impact both the social aspect of the course, in that students were able to quickly develop a peer group, and also their ability to problem solve. The benefits of our integrated approach to collaborative learning can be seen both in terms of increased student confidence and perception of ability, and in increased student ability to complete problem solving activities.

Of interest is whether students continue to remain engaged in their subsequent courses, and whether their confidence remains high. We are currently investigating, though surveys of student satisfaction in subsequent courses, whether there are any long term changes in student perception as a result of learning community support.

References

Allan, V. & Kolesar, M. (1997), 'Teaching computer science: A problem solving approach that works', *ACM SIGCUE Outlook* **25**(1-2), 2–10.

Barrows, H. (1986), 'A taxonomy of problem-based learning methods', *Medical Education* **20**, 481–486.

Beder, S. (1997), 'Addressing the issues of social and academic integration for first year students: A discussion paper', *Ultibase Articles* .

Biggs, J. & Tang, C. (2007), *Teaching for Quality Learning at University, 3rd edition*, The Society for Research into Higher Education.

Burns, R. (1991), 'Study and stress among first year students in an australian university', *Higher Education Research and Development* **16**, 61–77.

Cameron, H. (2007), Assessment and feedback in higher education: Key links in the learning chain, *in* 'Proceedings of the 2nd Education Research Group of Adelaide Conference (ERGA 2007)'.

Carr, W. & Kemmis, S. (1986), *Becoming Critical. Education, knowledge and action research*, Lewes: Falmer.

Christensen, C., Garvin, D. & Sweet, A. (1991), *Education for Judgement: The Artistry of Discussion Leadership*, Harvard University Business School.

D'Souza, D., Hamilton, M., Harland, J., Muir, P., Thevathayan, C. & Walker, C. (2008), Transforming learning of programming: a mentoring project, *in* 'Proc. Tenth Australasian Computing Education Conference (ACE 2008)', pp. 75–84.

Ericsson, K. A., Krampe, R. T. & Tesch-Romer, C. (1993), 'The role of deliberate practice in the acquisition of expert performance', *Psychological Review* **100**(3), 363–406.

Ginat, D. (2001), Misleading intuition in algorithmic problem solving, *in* 'Proceedings of the 32nd SIGCSE Symposium', pp. 21–25.

Ginat, D. (2008), Learning from wrong and creative algorithm design, *in* 'Proceedings of SIGCSE'08'.

Glasser, W. (1990), *The Quality School: Managing Students without Coercion, 1st edition*, New York: Harper Collins, ISBN: 006055200X.

Hopkins, D. (1985), *A teacher's guide to classroom research*, Philadelphia: Open University Press.

Ilian, A. & Kordaki, M. (2006), 'Undergraduate studies in computer science and engineering: gender issues', *ACM SIGCSE Bulletin* **38**(2), 81–85.

Jenkins, J. & Jenkins, L. (1987), 'Making peer tutoring work', *Educational Leadership* **44**(6), 64–68.

Krause, K. (2005), 'Serious thoughts about dropping out in first year: Trends, patterns and implications for higher education', *Studies in Learning, Evaluation, Innovation and Development* **2**(3), 55–67.

Krause, K., Hartley, R., James, R. & McInnis, C. (2005), *The First Year Experience in Australian Universities: Findings from a decade of national studies*, Canberra: DEST.

Lang, C., McKay, J. & Lewis, S. (2007), 'Seven factors that influence ict student achievement', *ACM SIGCSE Bulletin* **39**(3), 221–225.

Lister, R., Adams, E., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Mostrom, E., Sanders, K., Seppala, O., Simon, B. & Thomas, L. (2004), 'A multi-national study of reading and tracing skillls in novice programmers', *SIGCSE Bulletin* **36**(4), 119–150.

MacGregor, J. (1991), 'What difference do learning communities make?', *Washington Center News, Washington Center for Undergraduate Education* **6**(1).

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagen, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I. & Wiusz, T. (2001), 'A multi-national, multi-institutional study of assessment of programming skills of first-year cs students', *SIGCSE Bulletin* **33**(4), 125–140.

McInnis, C., James, R. & Hartley, R. (2000), Trends in the first year experience in australian universities, Technical report, DETYA, Canberra.

McKinney, D. & Denton, L. (2006), Developing collaborative skills early in the cs curriculum in a laboratory environment, *in* 'Proceedings of SIGCSE 2006, Symposium on Computer Science Education'.

Miliszewska, I. & Tan, G. (2007), 'Befriending computer programming: A proposed approach to teaching introductory programming', *Issues in Informing Science and Information Technology* **4**, 278–289.

Miller, A. & Kay, J. (2002), A mentor program in cs1, *in* 'Proceedings of ITiCSE 2002, Aarhus, Denmark'.

Ramsden, P. (2003), *Learning to Teach in Higher Education*, RoutledgeFalmer, London.

Sheard, J., Carbone, A., Markham, S., Hurst, A., Casey, D. & Avram, C. (2008), Performance and progression of first year ict students, *in* 'Proceedings of the Tenth Australasian Computing Education Conference (ACE 2008)'.

Slavin, R. (1983), 'When does cooperative learning increase student achievement?', *Psychological Bulletin* **94**(3), 429–445.

Slavin, R. (1991), 'Synthesis of research on cooperative learning', *Educational Leadership* pp. 71–82.

Smith, B. & MacGregor, J. (1992), *What is Collaborative Learning?*, National Centre on Postsecondary Teaching, Learning and Assessment, Pennsylvania State University.

Sonnentag, S. (1998), 'Expertise in professional software design: A process study', *Journal of Applied Psychology* **83**(5), 703–715.

Strazdins, P. (2008), Applying the community of practice approach to individual it projects, *in* 'Proceedings of the Tenth Australasian Computing Education Conference (ACE 2008)'.

Teague, D. & Roe, P. (2008), Collaborative learning - towards a solution for novice programmers, *in* 'Proceedings of the Tenth Australasian Computing Education Conference (ACE 2008)'.

Trajovski, G. (2006), *Diversity in Information Technology Education: Issues and Controversies*, Idea Group Inc (IGI).

Urban-Lurain, M. & Weinshank, D. J. (2000), Attendance and outcomes in a large, collaborative learning, performance assessment course, *in* 'Annual Meeting of the American Educational Research Association (AERA)'.

Webb, N. (1989), *Peer Interaction and Learning in Small Groups*, New York: Pergamon Press, pp. 21–29.

Wenger, E. (1998), 'Communities of practice. learning as a social system', *Systems Thinker* .

White, N. R. (2006), 'Tertiary education in the noughties: the student perspective', *Higher Education Research & Development* **25**(3), 231–246.

Williams, L. & Kessler, R. (2000), The effects of "pair pressure" and "pair-learning" on software engineering education, *in* 'Proceedings of the 13th Conference on Software Engineering Education & Training'.

Yerion, K. & Rinehart, J. (1995), 'Guidelines for collaborative learning in computer science', *ACM SIGCSE Bulletin* **27**(4), 29–34.

Losing Their Marbles: Syntax-Free Programming for Assessing Problem-Solving Skills

Colin Fidge and Donna Teague
Faculty of Information Technology,
Queensland University of Technology,
Brisbane, Australia
{c.fidge, d.teague}@qut.edu.au

Abstract

Novice programmers have difficulty developing an algorithmic solution while simultaneously obeying the syntactic constraints of the target programming language. To see how students fare in algorithmic problem solving when not burdened by syntax, we conducted an experiment in which a large class of beginning programmers were required to write a solution to a computational problem in structured English, as if instructing a child, without reference to program code at all. The students produced an unexpectedly wide range of correct, and attempted, solutions, some of which had not occurred to their teachers. We also found that many common programming errors were evident in the natural language algorithms, including failure to ensure loop termination, hardwiring of solutions, failure to properly initialise the computation, and use of unnecessary temporary variables, suggesting that these mistakes are caused by inexperience at thinking algorithmically, rather than difficulties in expressing solutions as program code.

Keywords: Learning to program; syntax-free programming; algorithms; problem solving.

1 Introduction

ITB001, *Problem Solving and Programming*, is the core introductory programming unit presented by the Faculty of Information Technology at the Queensland University of Technology. In it students are introduced to the skills required to solve computational problems and implement solutions in a programming language. Python is the language of choice for this unit, as it has a simple syntax and is easy to use. Students are not distracted with difficult installation tasks, nor do they need to master a complex development environment. As a freely-available scripting language, Python encourages experimentation by students to reinforce conceptual material (Zelle 2007).

Students of ITB001 are expected to devise a plain English "algorithmic" solution to each programming problem prior to implementation of their solution in program code. Stepwise refinement of abstract requirements to executable code is also encouraged, as a way of helping students develop programs incrementally. Marks awarded for assignments give almost as much weight to algorithms as programs, in an attempt to encourage methodical program development, rather than 'hacking'. In the teaching materials we used the term *prosecode*, borrowed from Zobel (2004), as a means of describing "syntax-free" algorithms, because we felt that most pseudo-code notations in the literature were still too "program-like" for our needs.

In Semester 1, 2008 we conducted an experiment involving 313 students sitting the final end of semester exam in this unit. Most of these students were first-year Information Technology students, although there was also a large number of second-year double-degree students. The unit assumes that students have no prior programming experience (although many students have some programming experience from high school or other university units).

While most of the exam questions emphasised traditional program code comprehension and writing skills, one exam question required students to write a natural language algorithm only, in order to test their ability to solve algorithmic problems independently of the specifics of any particular programming language. Here we examine the outcomes of this experiment by analysing the variety of algorithms produced by the students to see what they reveal about the students' underlying problem solving skills. We also compared the students' outcomes for this exercise in "syntax-free" programming against their solutions to a traditional program code writing exam question, to see whether students' skills at devising algorithms correlate strongly to their programming skills.

2 Related Work

Bornat (in Bruce et al. 2002) introduced the concept of "syntax-free" programming to separate the notions of *programming* from *coding* because of what he saw as the 'damage' caused by the introduction of a programming language while learning to program. Bornat clearly distinguished learning to program from learning to code in his book designed to teach programming outside an electronic environment. Our experiment tries to evaluate *problem solving* skills separately from *coding* in order to determine any correlation between the two.

Of course, many educators have used *"explain in plain English"* style questions to test students' program comprehension skills (Lister et al. 2006), but our experiment is different because it studies students' plain English *writing*, rather than reading skills.

Dierbach and others (2005) designed a preparatory course for novice programming students which sought to develop their ability for algorithmic problem solving rather than knowledge of any particular programming

Copyright (c)2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

language. They found that students in this course performed on average at a higher level in subsequent programming units than students with any other type of programming background. They also reported that the confidence levels of students completing the preparatory course were affected positively, and they suspect this contributed to their success in further programming courses. This supports the notion that confidence is a contributing factor to success and retention in programming courses (Fisher et al. 2002; Katz et al. 2006).

Others have suggested separating algorithm development from implementation of code and have drawn parallels between mastering algorithm development skills and successfully learning to program (DuHadway 2002; Faux 2003). Mendes et al (2005) suggested that a significant problem for students learning to program is their inability to develop an algorithm to solve a problem, which is one of our motivations for trying to study the two skill sets separately.

Reporting on a series of projects investigating what they termed 'common sense computing', Simon et al (2006) and Chen et al (2007) suggested that prior to commencement of computer science studies, students already have considerable problem solving and algorithm writing skills, but found that those skills actually deteriorate with fragile programming knowledge.

In terms of code implementation, Parsons and Haden (2006) observed that novice programmers repeatedly made the same syntactical errors. They developed an interactive puzzle-like tool where students drag-and-drop portions of code to form a complete solution. The puzzles allow students to identify errors they make in choosing inappropriate lines of code, and provide a fun way to rote learn syntactic constructs.

Finally, Little and Miller (2008) acknowledge the difficulties associated with language-specific syntax and are currently working towards syntax-free programming by developing a parser that generates code from a plain-English set of instructions. Such a utility, although not helpful to a student struggling to learn a specific language, could be useful in building problem solving and algorithm development skills.

3 The Question

The final exam question used in our experiment was as follows:

Assume that you have a five-year old niece to whom you have just given a paper bag full of marbles, each of which has a unique number printed on it. Write a detailed set of step-by-step instructions (i.e., an algorithm) for your niece to follow which will allow her to find the two marbles with the smallest and largest numbers in the bag, respectively.

You may assume that she can recognise whether or not one number is larger than another, but that she cannot understand vague instructions like "find the marble with the largest number". (Don't worry about whether or not your niece can read — an adult can read out the instructions to her if necessary.)

The style of this question should not have been a surprise to the students. Developing natural language algorithms before writing code was emphasised throughout the semester in both weekly workshop exercises and assessable assignments. During the semester students were expected to write an abstract English description of their problem solving strategy, prior to coding a solution. This was then repeatedly refined until a concise set of non-ambiguous instructions was developed which solved the task. Extensive illustrative examples documenting this step-wise refinement process from abstract algorithm to code were supplied to students each week.

Also a similar question involving counting playing cards by suit, to confirm the integrity of a second-hand deck, appeared on the practice exam presented in the final lecture. The ploy of requiring the algorithm to be written for a small child was used on the practice exam too, as a way of promoting the idea that each step in the algorithm must be very simple. A solution was provided for the corresponding practice exam question, clearly showing the style of answer expected. Students should have been quite familiar with the concept of collections (lists), as list processing, operations and comprehension formed a major part of the semester's work.

4 The Expected Answer

The answer we expected to the exam question above was something along the following lines:

1. *Draw two circles on a piece of paper and label one "largest" and the other "smallest".*
2. *Take two marbles from the bag and put the one with the smaller number in the circle labelled "smallest" and the other one in the circle labelled "largest".*
3. *While there is still a marble left in the bag:*
    a. *Take a marble from the bag.*
    b. *If the marble has a smaller number on it than the one in the circle labelled "smallest", put it in place of that marble in the circle.*
    c. *Otherwise, if the marble has a larger number on it than the one in the circle labelled "largest", put it in place of that marble in the circle.*
    d. *Otherwise, just put the marble aside.*
4. *When there are no more marbles left in the bag, the one left in the circle labelled "smallest" is the one with the smallest number and the one in the circle labelled "largest" is the one with the largest number.*

This set of instructions is a prosecode paraphrase of Berman and Paul's (2005) 'MaxMin2' algorithm for efficiently finding the largest and smallest numbers in an array.

Although independent of any particular programming language syntax, note that this algorithm has all the features of basic imperative program code. The numbering of the steps represents *sequential composition* of actions. Step 1 in effect *declares* two named *variables* and Step 2 initialises them via *assignment*. Step 3 represents pre-tested *iteration*. Steps 3b–3d model a *conditional statement*. Finally, Step 4 represents the action of *returning* the computation's result.

Also note that this particular solution implicitly relies on an *assumption* that there are at least two marbles in the bag (which seems reasonable given that the question says the bag is "full of marbles").

5 Solutions Produced

To analyse students' responses to this question we categorised them by the type of algorithm produced, by the assumptions their algorithm makes about its inputs (i.e., the minimum number of marbles required), and by how successfully the algorithm was expressed. We considered an attempt at describing an algorithm to be 'unsuccessful' if the instructions were impossible to follow because they were either wrong, incomprehensible or too vague for even an intelligent child to understand.

Overall, a surprisingly wide range of solutions were produced, some of which did not occur immediately to the teaching staff, and some of which actually improved on our solution by avoiding our assumption about the minimum number of marbles required. The anticipated solution described above is essentially a *searching* algorithm, and most students' answers were of this nature. However, a sizeable percentage of students produced solutions based on *sorting* the marbles, in order to find the smallest and largest ones. Although this strategy works, it is inefficient because there is no need to sort the marbles to solve the problem. (Nevertheless, efficiency is not a marking criterion for this unit, so students were not penalised for suggesting such solutions.) Ultimately the algorithms were divided into 13 distinct categories, as shown in Table 1.

Category	**% Students Attempting**	**% Students Unsuccessful**
Single pass search, minimum 2 marbles	35.5	3.2
Two pass search, minimum 2 marbles	17.9	1.9
Partial sort	9.5	1.9
Ill-defined total sort	9.3	9.3
Total insertion sort	6.4	3.5
Single pass search, minimum 1 marble	6.1	1.3
Recursive sieve	4.5	3.2
Digit based sort	3.2	3.2
Two pass search, Minimum 1 marble	2.2	0.6
Unique/Incomprehensible	2.2	1.6
Total bubble sort	1.3	1.3
No answer	1	1
Single pass search, Minimum 3 marbles	1	0.6

Table 1 Algorithm Categories

Below we discuss interesting aspects of each specific category of algorithm produced.

Single pass search, minimum of 2 marbles [111/313 = 35.5% of answers attempted; 10/313 = 3.2% deemed unsuccessful]. This was the expected algorithm, described in detail above, so it was pleasing to discover that it was by far the most common solution.

Two pass search, minimum of 2 marbles [56/313 = 17.9% of answers attempted; 6/313 = 1.9% deemed unsuccessful]. A less-efficient, but equally effective, variant of the expected algorithm, this solution involves searching through the bag of marbles twice, firstly to find the one with the smallest number and then to find the one with the largest number. Most students recognised that having found the marble with the smallest number in the first pass, this particular marble could be excluded from the second.

'Partial sort' [30/313 = 9.5% of answers attempted; 6/313 = 1.9% deemed unsuccessful]. This commonly-suggested algorithm piqued our interest because it is quite unlike any examples or exercises studied in the unit. It begins by selecting two marbles and placing them in a line with the smaller on the left. Each of the remaining marbles is then compared with the marbles at the ends of the line. If its number is smaller than the one on the far left it is placed further to the left. If its number is larger than the one on the far right it is placed further to the right. Otherwise, it is placed "in the middle". When the bag of marbles is emptied the ones with the smallest and largest numbers can be found at the extreme ends of the line. Intriguingly, this process produces a *partial* ordering of the marbles.

At one level the algorithm can be viewed as merely a slight variant of our expected one, with the additional feature that marbles that are not candidates for largest or smallest are retained rather than discarded. However, the interesting feature is that the degree of sorting achieved depends on how the student explained what was meant by putting a marble "in the middle". In some cases "the middle" was clearly intended to be an unordered pile. In others, however, it was said to be a "line" (sequence). In this latter case different sorting outcomes can be achieved by putting a marble destined for "the middle" immediately beside a marble at the end of the line versus putting it in the exact centre of the sequence. Both of these alternatives were commonly suggested. (Of course, none of these variants make any difference to the overall answer produced by the algorithm.)

Ill-defined total sort [29/313 = 9.3% of answers attempted, all deemed unsuccessful]. A large percentage of students recognised that the problem could be solved trivially if the marbles were sorted, but did not know how to achieve this. Their algorithms therefore said, in essence, "sort the marbles and then choose the ones at the end". All such algorithms were considered unacceptably imprecise.

Total insertion sort [20/313 = 6.4% of answers attempted; 11/313 = 3.5% deemed unsuccessful]. Many students attempted to describe an algorithm which involved totally sorting the marbles and then returning the ones at the ends. (Inevitably such an algorithm will be unnecessarily inefficient because there is no need to totally order the marbles to solve the problem.) Most of these solutions used an insertion sort-like algorithm. A line of marbles is maintained in left-to-right ascending order and each newly-selected marble is placed in its appropriate place by searching from the left until a marble with a larger number is encountered, in which case the new marble is placed to its left, or the end of the line is reached, in which case the new marble is placed at the far right.

Although this algorithm is straightforward, it is awkward to explain in precise step-by-step detail. Most students had difficulty describing the process for searching for the location at which to insert each new marble, which is why many attempts were deemed unsuccessful. (Sorting algorithms are considered too difficult for students in this unit, so it was surprising that so many students attempted this approach.)

Single pass search, minimum of 1 marble [19/313 = 6.1% of answers attempted; 4/313 = 1.3% deemed unsuccessful]. Whether by accident or design, a large number of students produced solutions which improved on our expected algorithm by requiring only one marble as a minimum. However, this considerably complicated their algorithms. Typically the first marble selected was placed in the container labelled "largest". The loop for processing the remaining marbles, if any, then needed an extra conditional statement to check whether or not the container labelled "smallest" is occupied. If not, then the newly-selected marble is placed in the container labelled "smallest" if its number is less than that of the marble in the container labelled "largest", otherwise the marble in the "largest" container must be moved to the "smallest" one, and the new marble is placed in the "largest" container. In effect, the first and second (if there is one) marbles taken from the bag must both be treated as special cases.

A weakness of this strategy is that if there is only one marble in the bag then the result returned has a "largest" outcome but no "smallest" one. However, five students managed to overcome even this problem by writing down (or in one case memorising) the numbers on the marbles, rather than using the marbles themselves as 'values'. This allowed them to initialise their algorithm by recording the number on the first marble as *both* "smallest" and "largest". This solution was the most efficient and general of all algorithms produced. (A sixth student recognised the problem and attempted to solve it by writing an initial value of zero for both the smallest and largest numbers, but failed to take into account that '0' may be a legitimate number on a marble.)

'Recursive sieve' [14/313 = 4.5% of answers attempted; 10/313 = 3.2% deemed unsuccessful]. This interesting group of, largely unsuccessful, attempts was again entirely unexpected by the teaching team. Each began with the same initial steps: "Take marbles out of the bag two at a time and put the one with the largest number in a pile on the right and the other one in a pile on the left".

This process will, of course, create a pile known to contain the marble with the smallest number, and another pile known to contain the marble with the largest one, but we still don't know which marble is which! Having thus painted themselves into a corner, the students' algorithms then differed depending on whether or not they saw a way out. Many students' attempts simply trailed off with a vague instruction to "keep going".

The best attempts recognised that a recursive problem had been created and attempted to explain the need to repeatedly apply the process to each of the piles. However, this was awkward to express without having named subroutines to call and, in any event, the whole process was far more complicated than our "five-year niece" could be expected to follow. Also, although recursive algorithm design had been covered in the unit, most of these algorithms failed to clearly identify a 'base case' to ensure termination.

Another obvious weakness of this approach, even when described relatively successfully, is that it relies on the bag containing an even number of marbles. None of the attempts explained what to do if the number of marbles in the bag was odd.

'Digit-based' sorting [10/313 = 3.2% of answers attempted, all deemed unsuccessful]. Perhaps the most disappointing group of attempts were those which tried to sort the marbles' numbers one digit at a time, e.g., hundreds, tens and units, or based on the number of digits the numbers contained. The reason for adopting such an awkward process, rather than just comparing the numbers on the marbles directly, was never explained. Some grouped the marbles by digits, or ranges of values, and then proceeded to perform a linear search through the groups anyway, making the initial grouping phase redundant.

Many of these algorithms were doomed to failure because a sufficient spread of numbers was not catered for. For instance, some said, "put all the marbles with a single digit in a pile and then select the smallest of these", ignoring the possibility that there may be no marbles with single-digit numbers.

All of these algorithms were inefficient, confusing and complex, and certainly beyond the capabilities of our "five-year niece".

Two pass search, minimum of 1 marble [7/313 = 2.2% of answers attempted; 2/313 = 0.6% deemed unsuccessful]. This algorithm was a variant of the 'two pass' algorithm outlined above, but took greater care to consider the possibility that there is only one marble in the bag. Again two searches are made through the collection of marbles, but the result of the first pass is written down and the chosen marble returned to the bag, thus allowing the same marble to also emerge from the second pass.

Unique and/or incomprehensible solutions [7/313 = 2.2% of answers attempted; 5/313 = 1.6% deemed unsuccessful]. A small number of attempts defied classification. Notable examples were as follows.

- One student began by dividing the marbles into two piles as described in the 'recursive sieve' algorithm above. The algorithm then proceeded to totally sort each of the piles before selecting the marbles of interest. (Although grossly inefficient, this strategy could succeed.)
- Another suggested sorting the marbles into piles of marbles whose number ends with the same digit and then searching through each of the piles

for the smallest and largest in that pile. (It was unclear how the algorithm was meant to terminate.)

- Another described in detail how to count the marbles but only briefly mentioned the issue of finding the ones with the largest and smallest numbers.
- Undoubtedly the strangest algorithm of all involved drawing a 10×10 matrix with rows and columns numbered with the digits from 0 to 9. (The numbers on the marbles were apparently assumed to all have two digits.) The marbles were then placed in the cells in the matrix according to their numbers' first and second digits. The assumption was then that the marble with the smallest number would be "closest" to the top left and the one with the largest number would be closest to the bottom right. Unfortunately, of course, there is nothing to prevent two or more marbles from being equidistant to the corners.

Total bubble sort [4/313 = 1.3% of answers attempted, all deemed unsuccessful]. A small number of students attempted a solution which involved sorting the marbles using a swapping-based algorithm similar to the traditional bubble sort. (Although sorting algorithms are not covered in the unit, a bubble sort was used as an illustration of an inefficient algorithm in a tutorial on complexity analysis, so the students may have had vague memories of this.) Unfortunately none of the attempts managed to describe the multiple passes needed to sort numbers in this way with sufficient clarity.

No answer [3/313 = 1.0% of students]. Only a few students failed to provide any answer at all to the algorithm question, unlike the code-oriented questions on the exam which was not attempted by 11 students. This might indicate that students sitting the exam considered the prosecode question to be easier than those involving programming language code.

Single pass search, minimum of 3 marbles [3/313 = 1% of answers attempted; 2/313 = 0.6% deemed unsuccessful]. This rarely-suggested approach is not a distinct algorithm, but a poorly-initialised version of our expected one. It begins by designating spaces for the "smallest", "largest" and "discarded" marbles. The next step then (unnecessarily) involves removing *three* marbles from the bag, to populate each of these three spaces. There is, of course, no need to initialise the "discarded" pile in this way.

5.1 Analysis of Students' Answers

Overall a clear majority of students produced a search-based solution as expected. Various linear searching algorithms for processing lists had been used throughout the semester as lecture examples, workshop exercises and assignment problems, so this outcome was not unexpected.

However, the large number of students who attempted some kind of (unnecessary) sorting algorithm was surprising. Sorting algorithms *per se* were not covered in this unit, apart from a lecture demonstration of an algorithm for separating negative and non-negative numbers from a list, and workshop exercises comparing algorithm efficiency using supplied sorting modules. Exposure to these modules was limited, with most students blindly calling the supplied code, so it is unlikely to have been particularly influential on students answering the exam question. One explanation for so many students using a sorting algorithm is, simply that knowledge of sorting is a pre-existing skill. Chen et al (2007) found in an experiment involving writing natural language solutions that prior to commencement of studies, students were able to articulate coherent versions of insertion, selection, bubble and other sorting algorithms.

Also, as mentioned above, many of the students' attempts surprised us. The 'partial sort' algorithms were a pleasant surprise because this is an elegant variant on the expected answer. However, the 'recursive sieve' algorithms sit at the other extreme. It is not clear why so many students adopted this obviously complex and awkward approach. We have observed that a small number of students work more confidently with recursive problems than even the simpler looping constructs of *for* and *while* statements. It is not unreasonable to expect that, under exam conditions, with virtually unlimited scope, students would choose an approach they were comfortable with. Although recursion is not now a huge part of the curriculum for this unit, in previous semesters the programming language used was Scheme, and recursion was the only tool available for expressing repetition. The experiment was conducted in a semester with a high percentage of repeating students who may have been exposed, at least to some degree, to the recursion-intensive predecessor introductory programming unit.

Similarly, the various 'digit-based' attempts were very disappointing. Students had previously completed a workshop exercise which required them to calculate the number of digits in a given integer, so their recall of this activity may have influenced their approach to solving the exam question. Other studies have similarly found that a large number of students treated numbers as strings of digits rather than as primitive types in order to describe a sorting algorithm and concluded that this was probably because they focused at school on the fact that digits were the basic parts of a number (Simon et al. 2006; Chen et al. 2007).

6 Coding Errors Evident in the Algorithms

Even though the algorithms were written in English, it was interesting to note that many novice programming errors could be discerned in them. Notable examples included the following.

Incomplete or incorrect initialisation. A handful of students failed to include a loop initialisation step in their algorithm and thus started attempting to compare newly-selected marbles without creating initial candidates for largest and smallest. A couple of students, apparently trying to allow for the possibility of there being only one marble in the bag, accidently created an 'uninitialised variable' problem by not having initial values for both the biggest and smallest numbers. More subtly, two students incorrectly initialised their single-pass search by discarding one of the first marbles taken from the bag, in a way that meant it was never considered as a candidate for either largest or smallest. (These could be difficult bugs to detect in the corresponding program!)

Inadequate declaration. Most students followed the lead of the algorithm on the practice exam and introduced clearly labelled 'variables' in which to put the selected marbles. Sometimes these were pieces of paper labelled "biggest" and "smallest" but, given the propensity of marbles to roll, most students had the foresight to suggest labelling bowls, bags or other containers. However, a small number of students just suggested allocating certain unlabelled 'places' to put the marbles. Since these anonymous 'places' had no names, the remainder of the student's instructions then struggled to distinguish them, introducing awkward phrases such as "the place where you put the biggest marble" or, in one memorable example, "space 1", "space 2" and "space 3", creating considerable scope for confusion. Evidently these students had not absorbed our frequent admonitions to choose meaningful variable names.

Unnecessary temporary variables. Several students introduced the equivalent of 'variables' into their programs that were entirely unnecessary. In a few cases students suggested writing down the largest and smallest numbers seen on marbles so far, rather than just retaining the marbles themselves. These students lost sight of the fact that this exercise was in essence to return the *marbles* with the smallest and largest numbers, not the *values* on those marbles.

Even stranger, two students suggested writing down the numbers from *all* the marbles on separate pieces of paper first and then searching through the pieces of paper instead of the marbles. Odder still, one student suggested counting all the marbles as the first step, even though this total was never used subsequently. Another student described a partial sort which involved writing down the number on every marble examined so far and, worse, searching through *all* these numbers for each newly-selected marble. (At the other extreme, only one student suggested using the niece's own memory to keep track of the smallest and largest numbers seen so far.)

Failure to ensure loop termination. A particularly obvious error in a few algorithms was failure to ensure that the loop made progress towards termination. In our situation this typically manifested itself as an instruction to return each marble examined to the bag from whence it came! (One student did say "choose another marble" from the bag after returning the marble just examined, but offered no specific advice on how to keep track of which ones had already been considered.)

Hardwiring of solutions. The standout cases in this category were the various 'digit-based' attempts, all of which suffered from a need to know the range of numbers written on the marbles. A few attempted solutions seemed to assume that the numbers were in an 'obvious' range, such as 1 to 100, although they didn't state this. Of course, the only real requirement for the numbers on the marbles is that they are totally ordered.

Failure to return a value. A minor issue with many algorithms is that they did not clearly say where the 'result', i.e., the marbles with smallest and largest numbers, could be found. This was not a problem when the marbles of interest were placed in clearly labelled containers, but some students chose to label their 'variables' with different colours, e.g., a red and a green bowl to hold the marbles selected.

7 Problem Solving Versus Coding Skills

Directly following the algorithm development question on the exam, the students were required to complete a traditional program coding question:

> *Define a Python function called* `twinned` *which accepts a list of items and returns a list of those items that occur exactly twice in the given list. Each such item may occur in the returned list once only. Hint: Python's* `count` *method for lists, which returns the number of times a particular item occurs in a list, simplifies the solution.*

Again, this question should not have been a surprise to the students as they were shown a similar example in a lecture, which tested if all items in a given list are unique. The solution to the question above is straightforward and involves iterating over the items in the given list and adding each one to a new list only if the item appears twice in the original list and not in the new one.

Students who answered this question generally did fairly well. The most common mistake was forgetting to ensure that the returned list of items does not contain duplicates.

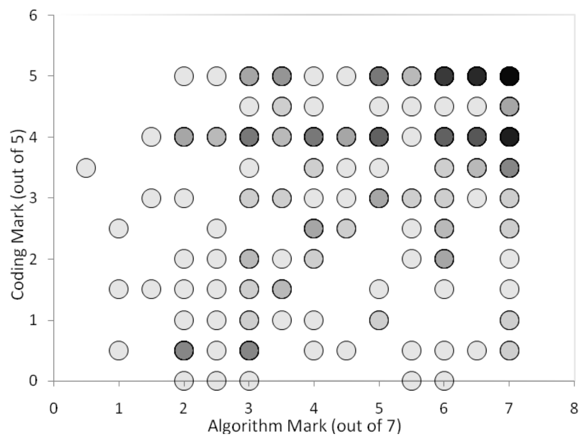


Figure 1: Student marks for algorithm and coding questions

For each of the students who sat the exam we thus had a marked "algorithm" and "coding" question, allowing us to do a direct correlation analysis between the two, to see if students' skills at writing algorithms translate to skills at writing program code. For this analysis we removed students who had failed to provide an answer to either or both of the exam questions. There are any number of reasons why questions go unanswered on exam papers, and no presumption can or has been made about the level of competence or otherwise of students providing little or no written clues. Two students answered neither question, and ten others failed to answer one of them, so this comparison focused on 301 student marks. A standard Pearson's correlation coefficient analysis, to produce a result ranging from −1 (inverse correlation) to +1 (positive correlation), for those 301 students produced a correlation of r = +0.446, showing little linear relationship between students' ability to answer the algorithm question and the coding question.

The scatter plot in Figure 1shows each student's mark for the algorithm question against their mark for the coding question. Since there are many duplicates, the density of each point is an indication of the number of students with the same algorithm and coding marks.

We also tried grouping the solutions to the algorithm question into the three main categories, searching algorithms, sorting algorithms, and the much smaller set of unusual solutions. Table 2 shows the distribution of marks within each category, including the mean and standard deviation, as well as the distribution of marks the same students received for the coding question.

	Algorithm Mark		**Coding Mark**	
Algorithm Category	**Mean**	**StDev**	**Mean**	**StDev**
Searching	5.7	1.6	3.7	1.4
Sorting	4.2	1.6	3	1.6
Other	2	1.4	2	1.9

Table 2 – Mark Distribution

The correlation coefficient, scatter plot (Figure 1) and mark distribution (Table 2) all show little correlation between students' ability to develop algorithms and to write code. Our original hypothesis was that students' algorithmic problem solving skills would be a strong indicator of their coding skills. The weakness of the correlation could be explained by two factors:

1. Some students who performed adequately on the algorithm question did poorly on the coding question. Although this is the most obvious explanation, this trend did not stand out dramatically during marking. (The same marker marked both questions.) Assignments gave similar weight to algorithms as code, and students were especially encouraged to concentrate on problem solving process and documenting their algorithm if they lacked the confidence or ability to produce working code. In the exam this type of student may have had ample experience at producing algorithms, but not programs, and therefore out-performed in the algorithm question. Another explanation is that those students who developed a habit of producing algorithms before code, finding no requirement to first produce an algorithm for the coding question in the exam, struggled to problem-solve directly to code.

2. Some students who did well on the coding question did poorly on the algorithm question. There were a few obvious cases of students who could write code but couldn't express themselves in English. One overseas student in particular achieved near full marks on the code-related questions but failed to write a comprehensible algorithm. Four students wrote Python code on the back of the page for the algorithm question! This is consistent with evidence that many students dislike documenting their problem-solving process and when forced to produce both algorithms and code for assessment, choose (contrary to instructions) to write the code first, then attempt to convert that into an algorithm. We suspect this to be the case more particularly for students with at least some prior programming experience.

Overall we believe the first of these explanations was the dominant one. It seems inevitable that many students who can express their solutions satisfactorily in plain language will have difficulty with the additional requirement to translate their algorithms into program code. The scatter plot is consistent with this explanation as evidenced by the slightly more dense concentration of data points in the bottom right hand corner than the top left.

8 Discussion

The way in which the exam question was worded, asking the students to direct their instructions towards a small child, worked far better than we expected. Previously we have always had difficulty explaining to students that we want algorithms expressed as a sequence of simple mechanical steps. Typically they produce monolithic paragraphs of confusing explanations, or steps that are far from "simple". The "five-year old niece" motivation seemed to solve this problem immediately. Although some algorithms were poorly explained, all of the exam question responses were closer to our ideal of an executable algorithm than most of their attempted algorithms in assignments.

Another pleasing side-effect of the experiment was the way in which students engaged with the spirit of the algorithm question. Many students included small asides in their solutions directed at the imaginary "five-year old niece". Some gave her a name, several advised her to spread the marbles out on carpet to prevent them rolling away, one warned her not to swallow the marbles, one promised her "cake" as a reward for finishing the algorithm, one used a picture of a mouse to indicate "smallest" and an elephant to indicate "largest", one said that the niece could write the labels needed for the algorithm in permanent pen on the kitchen bench "provided mummy is not around", and one even advised her not to take up drugs later in life! No such light-heartedness was evidenced in the more traditional programming questions.

It was also interesting to note that almost all students answered the "syntax free" question, unlike the code comprehension and writing questions, which many students failed to answer. Only three out of 313 students failed to provide any answer at all to the algorithm question, suggesting that students considered this an "easier" task. (Nevertheless, many students still performed badly on the question, despite generous marking.)

A practical disadvantage was the difficulty of marking the question, however. Given the wide range of responses, and the different forms of expression used by the students, each answer had to be read and assessed carefully. The algorithm question took twice as long to mark as the equivalent code writing question, which produced a more consistent style of response.

9 Conclusions

Successful computer programming involves two tasks, developing an algorithmic solution to the problem and then expressing this solution in the target programming language. To see if the first of these skills could be assessed separately from the second we conducted a large-scale experiment in which students' ability to write "syntax-free" algorithms and program code were both examined. We were surprised by the wide variety of solutions produced by the students, when freed of the requirement to express themselves in a specific programming language. It was also interesting to discover that many common coding errors (inadequate initialisation of variables, failure to guarantee loop termination, etc) could be seen clearly in the students' natural language algorithms. Nevertheless, the weaker than expected correlation between students' marks for the algorithm and coding questions means that a syntax-free programming question cannot be used as a complete substitute for a traditional code writing question.

10 Acknowledgements

We wish to thank our teaching colleagues, Fiona Cleary, John Hynd, Andrew Craik, Annabel Poppleton, Jason Wimmer, Matthew Brecknell, Neil Muspratt and Frederic Maire for their invaluable assistance and camaraderie throughout the semester in which this research was conducted.

11 References

Berman, K. A. and Paul, J. L. 2005. Algorithms: Sequential, Parallel and Distributed, Thomson.

Bruce, C. and McMahon, C. 2002. Contemporary Developments in Teaching and Learning Introductory Programming: Towards a Research Proposal. Faculty of Information Technology Teaching and Learning Report 2002 – 2. D. P. Bancroft. Brisbane, QUT.

Chen, T.-Y., Lewandowski, G., McCartney, R., Sanders, K. and Simon, B. 2007. Commonsense Computing: using student sorting abilities to improve instruction. Proceedings of the 38th SIGCSE technical symposium on Computer science education, Covington, Kentucky USA, ACM.

Dierbach, C., Taylor, B., Zhou, H. and Zimand, I. 2005. Experiences with a CS0 Course Targeted for CS1 Success 36th SIGCSE technical symposium on Computer science education St Louis, Missouri, USA, ACM.

DuHadway, L. P. 2002. Separating Fundamental Concepts from Language Syntax in an Introductory Computer Science Course. Logan, Utah, Utah State University.

Faux, R. J. 2003. Impact of a Pre-Programming Course in a Computer Science Curriculum, The Union Institute and University.

Fisher, A. and Margolis, J. 2002. "Unlocking the clubhouse: the Carnegie Mellon experience " ACM SIGCSE Bulletin 34(2).

Katz, S., Allbritton, D., Aronis, J., Wilson, C. and Soffa, M. L. 2006. "Gender, Achievement, and Persistence in an Undergraduate Computer Science Program." ACM SIGMIS Database 37(4).

Lister, R., Simon, B., Thompson, E., Whalley, J. and Prasad, C. 2006. Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. Eleventh Annual Conference on Innovation Technology in Computer Science Education (ITiCSE'06), Bologna, Italy, ACM.

Little, G. and Miller, R. 2008. "Syntax-free Programming." Retrieved 1/09/08, 2008, from http://groups.csail.mit.edu/uid/projects/keyword-commands/index.html.

Mendes, A. J., Gomes, A. and Esteves, M. 2005. "Using simulation and collaboration in CS1 and CS2." ACM SIGCSE Bulletin 37(3): 193-197.

Parsons, D. and Haden, P. 2006. "Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses." Eighth Australasian Computer Education Conference (ACE2006) 52: 157-163.

Simon, B., Chen, T.-Y., Lewandowski, G., McCartney, R. and Sanders, K. 2006. Commonsense Computing: What students know before we teach (Episode 1: Sorting). Proceedings of the 2006 international workshop on Computing education research.

Zelle, J. 2007. "Python as a first language." from http://mcsp.wartburg.edu/zelle/python/python-first.html.

Zobel, J. 2004. Writing for Computer Science. London, Springer.

Human Fallibility: How Well Do Human Markers Agree?

Debra Haley, Pete Thomas, Marian Petre, Anne De Roeck
The Centre for Research in Computing
The Open University
Walton Hall, Milton Keynes MK7 6AA UK

D.T.Haley, P.G.Thomas, M.Petre, A.DeRoeck at open.ac.uk

Abstract

Marker bias and inconsistency are widely seen as problems in the field of assessment. Various institutions have put in place a practice of second and even third marking to promote fairness. However, we were able to find very little evidence, rather than anecdotal reports, of human fallibility to justify the effort and expense of 2nd marking. This paper fills that gap by providing the results of a large-scale study that compared 5 human markers marking 18 different questions each with 50 student answers in the field of Computer Science. The study found that the human inter-rater reliability (IRR) ranged broadly both over a particular question and over the 18 questions. This paper uses the Gwet AC1 statistic to measure the inter-rater reliability of 5 markers.

The study was motivated by the desire to assess the accuracy of a computer assisted assessment (CAA) system we are developing. We claim that a CAA system does not need to be more accurate than human markers. Thus, we needed to quantify how accurate human markers are.

Keywords: assessment, marker reliability, marker bias, inter-rater reliability, Gwet AC1, computer assisted assessment.

1 Introduction and motivation for the study

Subsections 1.1 and 1.2 show that educators believe assessment is important and costly and that these two factors have led to increasing interest in Computer Assisted Assessment (CAA). One of the critical questions about CAA systems is: How do you measure the accuracy of a CAA system? We believe that a CAA system has *good enough* accuracy if its results agree with humans as well as humans agree with each other. Thus, it is necessary to have reliable figures on human inter-rater reliability (IRR). Although the literature makes claims about the lack of good human IRR, we have been unable to find evidence. This paper provides results of a study to determine human IRR; these results can be used when assessing the accuracy of a CAA system.

1.1 Importance of assessment

McAlpine (2002 p. 4) gives the following description of assessment:

Copyright © 2008, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

> " ...assessment is a form of communication. This communication can be to a variety of sources, to students (feedback on their learning), to the lecturer (feedback on their teaching), to the curriculum designer (feedback on the curriculum) to administrators (feedback on the use of resources) and to employers (quality of job applicants)."

Assessment is "a critical activity for all universities" (Conole & Bull, 2002 pp. 13-14) and "there is no doubt" about its importance (Brown, Bull & Pendlebury, 1997 p. 7). Assessment is "widely regarded as the most critical element of learning" (Warburton & Conole, 2003). One researcher claimed "… the most important thing we do for our students is to assess their work" (Race, 1995). One reason for the importance of assessment given by several researchers is that assessment can have a strong effect on student learning (Brown, Bull & Pendlebury, 1997; Berglund, 1999 p. 364; Daniels, Berglund, Pears & Fincher, 2004). Brown, Bull & Pendlebury (1997 p. 7) claimed students learn best with frequent assessment and rapid feedback and added that one reason assessment is so important is that the right type of assessment can lead to deeper learning (1997 p. 24).

1.2 The growth of interest in Computer Assisted Assessment (CAA)

Computer Assisted Assessment (CAA) is assessment delivered and/or marked with the aid of computers (Conole & Bull, 2002). A 2002 study reported an increasing interest in and use of CAA in the preceding five years (Bull, Conole, Davis, White, Danson & Sclater, 2002). The number of papers published at the annual CAA conferences at Loughborough University supports the 2002 study. The number has grown from 20 in 1999 (the third year of the conference and the first year for which figures are available) to 40 in 2007 (www.caaconference.com) with an average of about 37 papers a year.

Brown, Bull & Pendlebury (1997 p. 40) claimed that the increased interest in assessment in the previous ten years "arises from the [British] government's pincer movement of insisting upon 'quality' while at the same time reducing unit costs" and predict "further cuts in resources"; they claim a 63% cut in per student resources since 1973 (1997 p. 255).

Ricketts & Wilks (2002 p. 312) agreed with Brown, Bull & Pendlebury (1997) for the increasing interest in CAA – decreasing resources per student require a cost

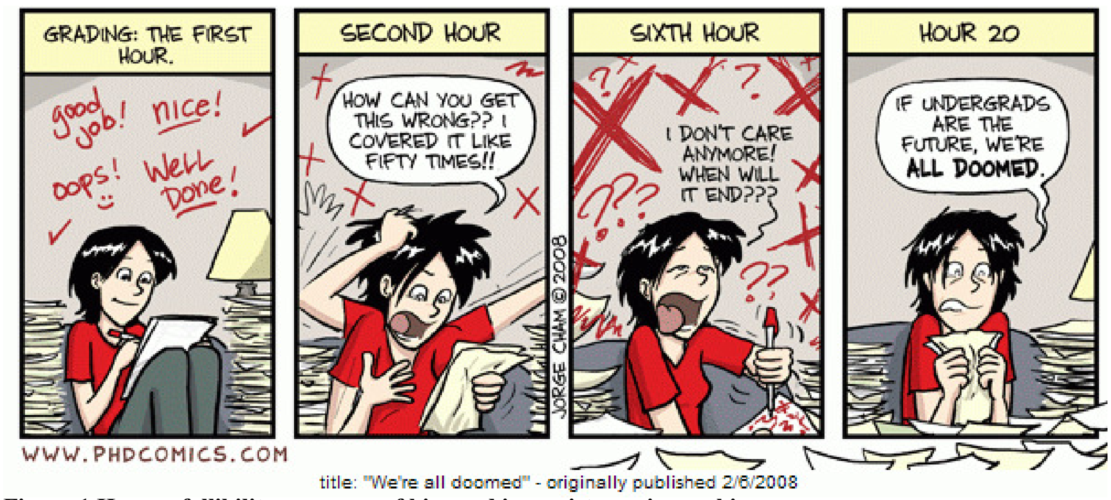

GRADING: THE FIRST HOUR.
good job!
nice!
oops!
Well Done!
SECOND HOUR
HOW CAN YOU GET THIS WRONG?? I COVERED IT LIKE FIFTY TIMES!!
SIXTH HOUR
I DON'T CARE ANYMORE! WHEN WILL IT END???
HOUR 20
IF UNDERGRADS ARE THE FUTURE, WE'RE ALL DOOMED.
JORGE CHAM © 2008
WWW.PHDCOMICS.COM
title: "We're all doomed" - originally published 2/6/2008

Figure 1 Human fallibility: a source of bias and inconsistency in marking

Used by permission: "Piled Higher and Deeper" by Jorge Cham www.phdcomics.com http://www.phdcomics.com/comics/archive.php?comicid=974

savings, which can be gained by decreasing tutor marking time. A 2003 survey (Carter, Ala-Mutka, Fuller, Dick, English, Fone & Sheard) gave a related reason for the interest in CAA: increasing enrolment. They cited the increasing number of ITiCSE (Integrating Technology into Computer Science Education) papers as evidence for the increased interest in CAA.

1.3 Reduce marker bias and improve consistency

In addition to the expected cost-savings, one goal of using CAA is to reduce marker bias and improve consistency. This subsection provides evidence that marker bias and inconsistency is *perceived* as a problem. Sections 3 and 4 provide *evidence* of marker inconsistency.

The papers cited used the terms bias and consistency without defining them. In the following paragraphs, we assume that bias is a prejudice either for or against a student and that consistency is a broader term referring to repeatability of results that can vary due to either bias or human error (e.g. adding marks or transcribing incorrectly, or differing judgments).

Figure 1 is a humorous depiction of how human fallibility can cause marker bias and lack of consistency.

Christie (2003) gave a comprehensive list of causes leading to lack of consistency. (Although Christie mentions essays, his comments generalize to short answers, which is the focus of this paper.) The comic strip exemplifies some of these factors.

"Manual marking is prone to several adverse subjective factors, such as:

- The length of each essay,
- The size of the essay set,
- The essay's place in the sequence of the essays being marked,
- The quality of the last few essays marked affecting the mark awarded to the essay currently being marked,
- The effect of the essayist's vocabulary and errors (spelling and grammar) on the marker,
- The marker's mood at the time of marking
- Marker's expectations of the essay set and of each essayist."

A thoughtful paper discussing a survey on bias (Sabar, 2002) reported that educators employ a wide range of solutions to the problem of how to resolve assessment difficulties arising from favouritism, implicitly acknowledging the ubiquity of possible bias in marking.

One study found bias in manual marking due to "inter-tutorial or intra-tutorial marking variations" (Summons, Coldwell, Henskens & Bruff, 1997). They claimed that reducing bias would have been "extremely difficult" without their CAA due to the large number of tutors and that most of their tutors "would have varied from the marking scheme". Thus, CAA led to more consistent marking.

The developers of a CAA system named Ceilidh (Benford, Burke, Foxley & Higgins, 1996) reported increased consistency using their CAA:

> "… hand marking of any form of coursework can lead to a student being treated less fairly than others. For instance, coursework marked by more than one person will lead to inconsistencies in marks awarded due to differing ideas of what the correct answer should be. This coupled with other problems such as racism, sexism and favouritism can lead to certain students achieving poorer marks than they deserve. We believe that such explicit discrimination is reduced, if not eliminated, by the use of the Ceilidh system since it marks each solution consistently."

Joy & Luck (1998) claimed that CAA provides consistency in marking: "… while the accuracy of marking, and consequently the confidence enjoyed by the students in the marking process, is improved. In addition, consistency is improved, especially if more than one person is involved in the marking process." Three years later, the consistency argument was still being made (Davies, 2001). An international survey (Carter, Ala-Mutka, Fuller, Dick, English, Fone & Sheard, 2003) reported that CAA is widely perceived to increase consistency in marking. Conole & Warburton agree with the survey that CAA "offers consistency in marking" (2005 p. 26). Tsintsifas (2002 p. 19) states:

> "Reliability and fairness increase by automating the assessment process because the same marking mechanism is employed to mark each piece of work. There is no possibility of discrimination and students are well aware of the fact that everyone is treated equally by the system."

The Open University (OU) follows formal procedures to address marker bias and inconsistency. We are particularly susceptible to these problems given the huge number of students and tutors involved in every presentation of a course. For example, almost 3,000 students took the computing course that this study used for data.

Part of the work involved in preparing a course is producing detailed Tutor Notes and Marking Schemes to help ensure marking consistency. Every exam undergoes moderation, that is, trained markers re-mark the exams and conflicting marks are investigated and resolved. A sample of all homework assignments is monitored to verify accuracy and consistency. These procedures are implicit evidence that OU believes human marking can suffer from bias and inconsistency.

This subsection gave examples of the widespread perception that human marking suffers from a lack of consistency. This perception, however, seems to be unsupported by empirical evidence and leads to the motivation for the study.

1.4 Motivation for the study

The papers cited in this subsection claimed, but did not provide evidence, that CAA improves marking consistency. Brown, Bull & Pendlebury (1997 p. 234) cite literature on general assessor inconsistency from 1890 to 1963. Newstead (2002), in an update of the classic article on the reliability of markers (Newstead & Dennis, 1994) provides evidence of poor marker reliability in the field of psychology. Despite these examples, we could find no literature that backed up, with evidence, the claim that CAA improves marking consistency in the field of computer science. To do so, the researchers would need to present evidence that human markers are not consistent either with each other and/or with themselves over time and that using CAA leads to improvement. This paper provides evidence that human markers are far from consistent, at least when marking short answers in the domain of computer science.

2 The Study

This section describes a study to evaluate how closely human markers agree with each other. It was part of a larger effort to develop a Computer Assisted Assessment system (CAA) to mark short answers in the domain of computer science.

2.1 The purpose of the study

A Computer Assisted Assessment system (CAA) is *good enough* if it agrees with human markers as well as human markers agree with each other. Thus, in order to evaluate our CAA, we needed to quantify how well human markers agree with each other. While it is often claimed that marking variability exists (see the introduction), it is difficult to find supporting evidence. This study provides evidence to support the claim that there is wide variability with human markers.

One can use the results of this study as a baseline against which to compare any CAA. If the results of a CAA closely match or exceed the baseline, then one can be assured that the CAA is *good enough*.

Inter-rater reliability (IRR) is the technical term used to describe how closely raters agree with each other. Gwet (2001 p. vii) states "Virtually anything that is used to generate explicitly or implicitly a measure for classifying a subject into a predefined category can be considered as a rater." He uses nurses diagnosing psychiatric patients (2001 p. 53) and scientists classifying fish according to colour (2001 p. 98) as examples of raters. In this paper, the raters are human markers. The subjects, analogous to Gwet's patients or fish, are student answers. The AC1 statistic was created to establish the level of agreement among raters (Gwet, 2001 p. vii).

2.2 The participants

We recruited five expert markers from the Open University (OU) staff. They have an average of 7.5 years experience as markers at the OU with an average of 3.5 years experience marking for the course from which we took the answers-to-be-marked. OU markers are highly trained – they go through a training course, mark to a detailed marking scheme, and are accustomed to having their marks moderated. As a sign of their conscientiousness, they often use a course on-line bulletin board to discuss intricacies of marking particular questions.

The reader should note that the marks collected for this study are un-moderated, that is, they were not checked, verified, and re-marked in the event of a disagreement between markers. Had the marks been intended for actual marking, they would have been moderated. Because OU courses can have thousands of students, it is customary for multiple markers to mark one course. The OU has procedures in place, including moderating marks and double-marking for high stakes assessments, to ensure a high level of consistency.

2.3 The Data

We used 18 different questions for this study (see Appendix A for the text of the questions). There are several types of questions; however, they are all from the first two homework assignments of the February 2004 presentation of M150 – Data, Computing and Information, which is an introductory course offered by OU's Computing Department. Some of the questions (e.g. 13, 14, 16) require quite concise, short, straight-forward answers while others (e.g. 4, 20) require longer, more open-ended answers. Some (e.g. 1 and 2) are multi-part and worth 8 and 12 points respectively while others are worth just 2, 3, or 4 points. Five questions (8-12) are about html. Thus, there is a variety of question types, although the main point is that they are all short answer, rather than multiple choice or true/false type questions.

Appendix A shows the text of the 18 questions for which the human markers evaluated the student answers. (Note that the 18 questions are numbered 1 to 21. Recall that the human marker study was part of a larger effort to develop a CAA system. We removed questions 5, 6, and 7 from the study because being numerical rather than textual, they were unsuited for marking by our assessment system.)

The student answers-being-marked came from the actual student scripts to questions given in the introductory computer literacy course mentioned above. Each of the five markers (with exceptions noted below) marked the same set of 60 random student answers to the 18 questions using the marking scheme created for the presentation of the course used in this study. We discarded the marks for the first 10 answers to each question so that the markers could become familiar with the marking scheme before we recorded their marks. To calculate the IRR of the five markers, we paired each of them with the other four for a total of ten human to human comparisons (markers 1 and 2, 1 and 3, and so on). These individual comparisons give an idea of the range of variation in human marking on these questions.

2.4 Validity

The study has good validity for several reasons. First, the participants were expert markers experienced in exactly the type of marking required by the study. In addition, the 18 questions were designed for an actual course presentation with no previous knowledge that they would be used to test the accuracy of human markers. The 50 answers marked for each question were genuine student answers. Finally, the large quantity of authentic data provides reassurance that the results can be generalised.

However, there are four possible threats to the validity of this study. One threat is the motivation of the markers, who were guaranteed anonymity and were paid for their work. Thus, if they were interested in completing the job as quickly as possible, they could have been careless with their marking. Unfortunately, we have no way of gauging the likelihood of this occurrence. This situation is somewhat analogous to real marking - markers are paid for their work. However, the guaranteed anonymity removed one reason for conscientious marking – in real marking situations, markers are monitored and one who consistently mismarks would not be rehired.

The second threat to validity is that the web interface between the markers and the marks database prevented the markers from reviewing their marks to adjust them, unlike their normal marking procedures. This could have resulted in less consistency than normal due to the inability of the markers to double-check their work. However, at least two of the markers were conscientious enough to *want* to review their marks. This fact may counterbalance the threat in the previous paragraph - that markers may have been careless because they were guaranteed anonymity.

The third point is that the results obtained from this study might show an unusually high level of agreement because all of the markers are experienced. Less experienced markers might not be as consistent as these markers. OU markers have years of experience carefully following a marking scheme to produce justifiably correct marks. In short, OU markers are good. Less experienced or less well-trained markers might not do as well.

Finally, due to a database overflow problem, two of the markers were unable to complete all of the marking. Thus, Question 17 was marked by just four humans and Questions 19-21 were marked by only three humans. Although this problem does not invalidate the results, it does mean that different questions have differing number of markers requiring care to be taken when comparing the results for the affected questions. However, one of the strengths of this study, the vast amount of data collected and analysed, still holds.

Despite the four problems mentioned in the previous paragraphs, we believe the study provides valuable results. The markers were professional and experienced (in contrast to many studies e.g. (Foltz, 1996) which use graduate students as markers), and the variety and authenticity of the questions as well as the expertise of the markers support the generalise-ability of the findings.

3 The results

Figures 1 through 18 in Appendix B display, for each of the 18 questions, the IRR using Gwet's AC1 statistic. For this metric, a higher AC1 number indicates that the relevant markers are closer in agreement than those with a lower AC1 number. Questions 1-16 and 18 were marked by five humans yielding ten pairs for each question. Question 17 was marked by four humans resulting in six pairs. Questions 19-21 were marked by three humans giving three pairs for each question.

In addition to calculating the IRR for each pair of markers, we calculated the overall IRR for all five markers (four for question 17 and three for questions 19-21). In each of the 18 figures, the horizontal line is the IRR for all of the markers; the segmented line shows the IRR for each pair of markers.

Figure 19 summarises the previous 18 figures; it shows the average IRR for each of the questions sorted from worst to best. This graph shows a wide range of values, from a low of 0.15 to a high of 0.97. The average IRR is 0.59 with a standard deviation of 0.27. By inspecting this figure, one can determine which questions show better agreement. Q19 shows the highest level of agreement while Q17 show the lowest level of agreement.

4 Discussion and implications

By glancing at the first 18 figures, one can see that for many of the questions, there is a large amount of inconsistency in the IRR figures within a single question. Questions 3, 4, and 15 show dramatic differences among the pairs of markers. For example, in Q4 the IRR ranges from a low of 0.01 for pair 1 and 4 to a high of 0.89 for pair 2 and 3. The average IRR for Q4 is 0.34. Seven pairs of markers were below this average and three pairs were substantially above the average.

In contrast to the questions with a wide variability in marking, in each of Questions 2, 13, and 16, the marker pairs are similar. For Q16, for example, the IRR ranges from 0.89 for pairs 1 and 4 and 4 and 5 to a high of 0.96 for pair 2 and 3; these ten pairs of markers have an average IRR of 0.92. These data suggest that Q16 is easy for human markers to mark at a high level of consistency.

For some of the questions, a particular marker or markers seem to lower the average IRR. For Questions 2, 3, 12, and 16, the worst four pairs contain marker 4; for Question 11, the worst four pairs contain marker 1, and for Question 15, the worst pairs contain marker 5. This observation has ramifications for evaluating the accuracy of a CAA system. If an observer can identify the CAA system as giving the least consistent marks, then one might conclude that the CAA system is not an adequate marker.

Figure 19 shows the average IRR for all of the 18 questions. They range from a low of 0.15 to a high of 0.97 with an average of 0.59. This huge difference from the lowest IRR to the highest IRR has a couple of implications. First, these data suggest that some questions are harder to mark than others. This difficulty could arise from an ambiguity in the question or a difference of opinion in how the marking scheme should be interpreted. Second, is the implication for the evaluation of a CAA system. Because the level of agreement among human markers depends on which question is being considered, it is necessary to compare the CAA system's marks and human IRR figures for one question at a time. An inaccurate impression of the accuracy of an automatic marker would be given if, for example, one reported that the average human IRR was 0.59 and the CAA achieved 0.57. The results of this study show that these two figures would overstate the CAA system's level of agreement with human markers for some questions and understate it for others.

5 Summary

The purpose of this study was to quantify how well human markers agree with one another in order to evaluate Computer Assisted Assessment Systems. By using Gwet's AC1 measure of inter-rater reliability, the study provides evidence that even very experienced and well trained markers often produce a wide range of IRR, both for the same question as well as for different questions.

The major conclusion from these data is that evaluating IRR is complex. It is not sufficient to report a single IRR figure. To gain a deeper understanding of the performance of raters, including automatic, computer-based raters, one needs to know the range and type of questions being marked as well as the IRR for each question.

6 References

Benford, S. D., Burke, E. K., Foxley, E. & Higgins, C. A. (1996). *Ceilidh: A Courseware System for the Assessment and Administration of Computer Programming Courses in Higher Education.* Nottingham, UK, The University of Nottingham, http://cs.joensuu.fi/~mtuki/www_clce.270296/Burke.html, last accessed 24 October 2007.

Berglund, Anders (1999). *Changing Study Habits - a Study of the Effects of Non-traditional Assessment Methods. Work-in-Progress Report.* **6th Improving Student Learning Symposium**, Brighton, UK.

Brown, G., Bull, J. & Pendlebury, M. (1997). **Assessing student learning in higher education**. London, Routledge.

Bull, Joanna, Conole, Grainne, Davis, H. C., White, Su, Danson, Myles & Sclater, Niall (2002). *Rethinking Assessment through Learning Technologies*. **Proceedings of ASCILITE 2002**, Auckland, New Zealand.

Carter, Janet, Ala-Mutka, Kirsti, Fuller, Ursula, Dick, Martin, English, John, Fone, William & Sheard, Judy (2003). *How Shall We Assess This?* **Proceedings of the ITiCSE 2003 working group reports**, Thessaloniki, Greece, ACM Press.

Christie, James (2003). *Automated essay marking for content - does it work?* **Proceedings of the 7th International CAA Conference**, Loughborough, UK.

Conole, Grainne & Bull, Joanna (2002). *Pebbles in the Pond: Evaluation of the CAA Centre*. **Proceedings of the 6th International CAA Conference**, Loughborough, UK.

Daniels, Mats, Berglund, Anders, Pears, Arnold & Fincher, Sally (2004). *Five Myths of Assessment*. **6th Australasian Computing Education Conference (ACE2004)**, Dunedin, New Zealand.

Davies, Phil (2001). *CAA must be more than multiple-choice tests for it to be academically credible?* **Proceedings of the 5th International CAA Conference**, Loughborough, UK.

Foltz, Peter W. (1996). *Latent semantic analysis for text-based research.* **Behavior Research Methods, Instruments and Computers 28**(2): 197-202.

Gwet, Kilem (2001). **Handbook of Inter-Rater Reliability: How to Estimate the Level of Agreement Between Two or Multiple Raters**. Gaithersburg, MD, STATAXIS Publishing Company.

Joy, Mike & Luck, Michael (1998). *Effective Electronic Marking for On-line Assessment*. **Proceedings of ITiCSE'98**, Dublin, Ireland.

McAlpine, Mhairi (2002). *Principles of Assessment.* CAA Centre, University of Luton, www.caacentre.ac.uk/resources/bluepapers/index.shtml, last accessed 28 October 2007.

Newstead, Stephen (2002). *Examining the examiners: Why are we so bad at assessing students?* **Psychology Learning and Teaching 2**(2): 70-75.

Newstead, Stephen & Dennis, I. (1994). *Examiners examined: The reliability of exam marking in psychology*. **The Psychologist 7**: 216-219.

Race, Phil (1995). *The Art of Assessing*. **New Academic 5**(3).

Ricketts, Chris & Wilks, Sally (2002). *What Factors affect Student Opinions of Computer-Assisted Assessment*. **Proceedings of the 6th CAA Conference**, Loughborough, UK.

Sabar, Naama (2002). *Towards principle practice in evaluation: learning from instructors' dilemmas in evaluating graduate students*. **Studies in Educational Evaluation 28**(4): 329-345.

Summons, Peter, Coldwell, Jo, Henskens, Frans & Bruff, Christine (1997). *Automating Assessment and Marking of Spreadsheet Concepts*. **Proceedings of the 2nd Australian Conference on Computer Science Education, SIGCSE**, Melbourne, Australia, ACM.

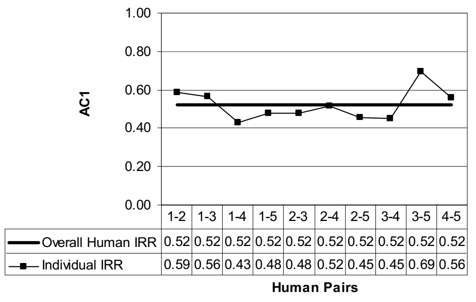
Tsintsifas, Athanasios. 2002. *A Framework for the Computer Based Assessment of Diagram Based Coursework*. unpublished PhD thesis. School of Computer Science and Information Technology, University of Nottingham, Nottingham. 235 pp.

Warburton, Bill & Conole, Grainne (2003). *Key Findings from recent literature on Computer-aided Assessment*. **Proceedings of ALT-C 2003**, Sheffield, UK.

Appendix A

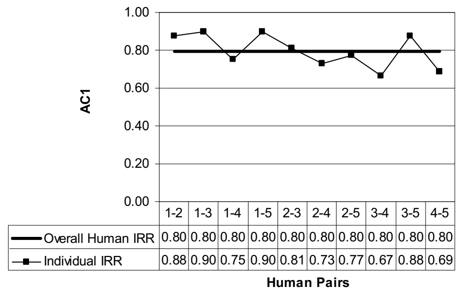
	Question Text	points
Q1	Name 2 elements of the course materials that will be distributed via the M150 course website?	8
	What is the role of the Study Calendar? What is the cut-off date for TMA02?	
	Find the learning outcomes for M150 which are listed in both the Course Companion and the Course Guide. Write down the learning outcome that you feel you are most interested in achieving and one or two sentences to describe why you have chosen that learning outcome.	
	What does eTMA stand for? What is the name of the document you should read to prepare yourself for submitting an eTMA? Who should you contact with queries about course software?	
Q2	Find the UK AltaVista site. What is its URI? What is the name of the large aquarium in Hull?	12
	Which query led you to the answer? What is the URI of the site?	
	What is the minimum number of intervening web pages you have to visit between the main site and the page that contains the information on the ballan wrasse?	
	List the URI of each intervening web page. How big can a ballan wrasse grow?	
	Does the ballan wrasse page tell you anything about the age a ballan wrasse can reach?	
	What age can a ballan wrasse reach?	
	What is the URI of the web page where you found the information?	
	Which search engine, and which query got you to the page that contained your answer?	
Q3	Explain, with examples, the difference between an analogue and a discrete quantity.	4
Q4	Give an example of a computer standard, explaining its purpose. Why is there a general need for standards in computing?	4
8-12	For each case; write the correct HTML and write one or two sentences about the problem with the original HTML. (The first line is the original HTML. The second line is the desired appearance.)	
Q8	<B>Always look left and right before crossing the road.	4
	Always look left and right before crossing the road.	
Q9	<B>Important!<B>Do <B> not place metal items in the microwave.	4
	Important! Do **not** place metal items in the microwave.	
Q10	<I>It is <B>very</I> </B> important to read this text carefully.	4
	It is ***very*** important to read this text carefully.	
Q11	Things to do: Things to do:	4
	Pack suitcase,<BR></BR>	
	Book taxi. Pack suitcase,	
	Book taxi.	
Q12	More information can be found <a name="help.htm">here</a>.	4
	More information can be found here.	
13-21	Victoria uses her computer to write up a report. When complete, she saves it to the hard disk on her computer. Later she revises her report and saves the final version with the same document name.	
Q13	Considering the contents of the report as data, at what point does the data become persistent?	2
Q14	What happens to the first saved version of the document?	2
Q15	Suggest an improvement in Victoria's work practice, giving a reason for your answer.	2
Q16	Give two examples of persistent storage media other than the hard disk.	2
Q17	Victoria then wishes to email a copy of her report, which includes data on identifiable individuals, to John, a work colleague at her company's Birmingham office. Write two sentences to explain the circumstances under which, within UK law, she may send the report.	2
Q18	Explain briefly the property of internet email that allows the contents of the report to be sent as an attachment rather than as text in the body of the email message.	2
Q19	John's email address is John@Birmingham.office.xy.uk Which parts of the address are: the user name, the name of the domain, the top-level domain?	2
Q20	Victoria then prepares her report for publication on a website. In no more than 100 words, explain what she has to take into account when making her report public.	3
Q21	Which of the following should she publish on the website with her report and why? Company address, personal telephone number, email address	3

Appendix B



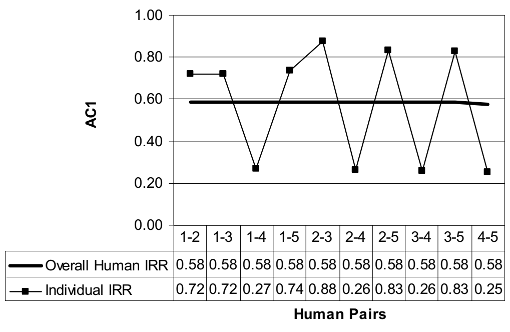
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.52	0.52	0.52	0.52	0.52	0.52	0.52	0.52	0.52	0.52
Individual IRR	0.59	0.56	0.43	0.48	0.48	0.52	0.45	0.45	0.69	0.56

Figure 1 Inter-rater Reliability for Question 1



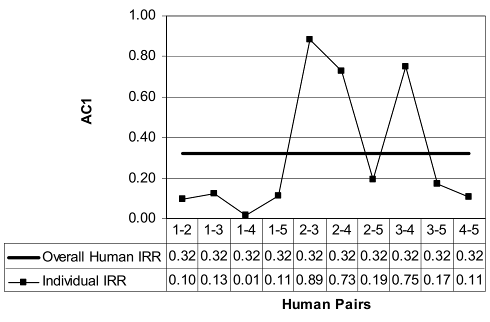
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.80	0.80	0.80	0.80	0.80	0.80	0.80	0.80	0.80	0.80
Individual IRR	0.88	0.90	0.75	0.90	0.81	0.73	0.77	0.67	0.88	0.69

Figure 2 Inter-rater Reliability for Question 2



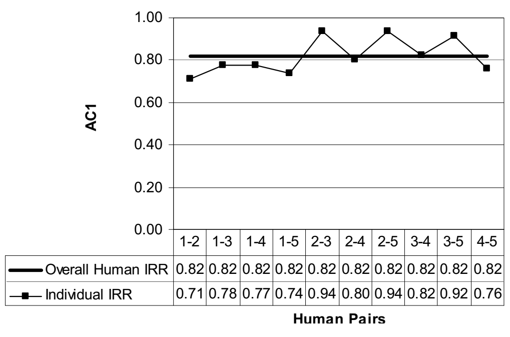
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.58	0.58	0.58	0.58	0.58	0.58	0.58	0.58	0.58	0.58
Individual IRR	0.72	0.72	0.27	0.74	0.88	0.26	0.83	0.26	0.83	0.25

Figure 3 Inter-Rater reliability for Question 3



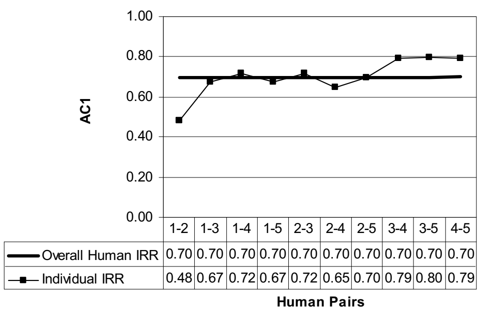
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.32	0.32	0.32	0.32	0.32	0.32	0.32	0.32	0.32	0.32
Individual IRR	0.10	0.13	0.01	0.11	0.89	0.73	0.19	0.75	0.17	0.11

Figure 4 Inter-rater Reliability for Question 4



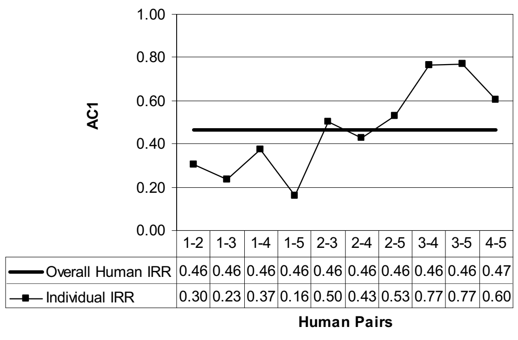
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.82	0.82	0.82	0.82	0.82	0.82	0.82	0.82	0.82	0.82
Individual IRR	0.71	0.78	0.77	0.74	0.94	0.80	0.94	0.82	0.92	0.76

Figure 5 Inter-rater Reliability for Question 8



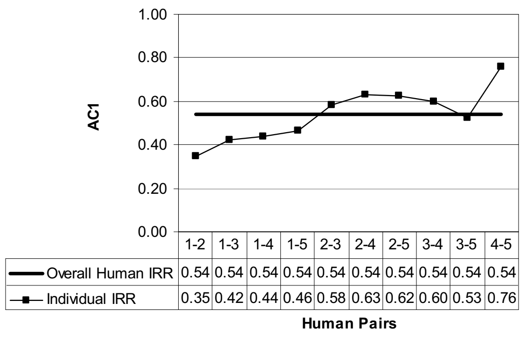
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.70	0.70	0.70	0.70	0.70	0.70	0.70	0.70	0.70	0.70
Individual IRR	0.48	0.67	0.72	0.67	0.72	0.65	0.70	0.79	0.80	0.79

Figure 6 Inter-rater Reliability for Question 9



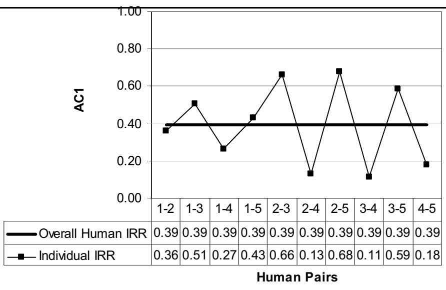
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.46	0.46	0.46	0.46	0.46	0.46	0.46	0.46	0.46	0.47
Individual IRR	0.30	0.23	0.37	0.16	0.50	0.43	0.53	0.77	0.77	0.60

Figure 7 Inter-rater Reliability for Question 10



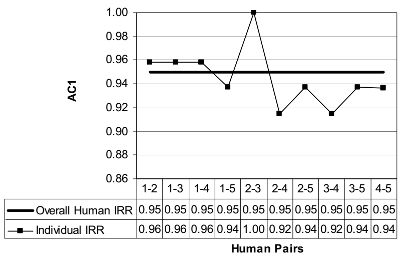
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.54	0.54	0.54	0.54	0.54	0.54	0.54	0.54	0.54	0.54
Individual IRR	0.35	0.42	0.44	0.46	0.58	0.63	0.62	0.60	0.53	0.76

Figure 8 Inter-rater Reliability for Question 11



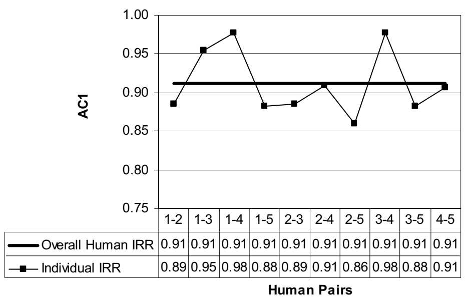
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.39	0.39	0.39	0.39	0.39	0.39	0.39	0.39	0.39	0.39
Individual IRR	0.36	0.51	0.27	0.43	0.66	0.13	0.68	0.11	0.59	0.18

Figure 9 Inter-rater Reliability for Question 12



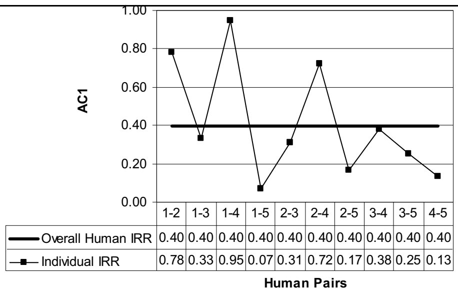
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95	0.95
Individual IRR	0.96	0.96	0.96	0.94	1.00	0.92	0.94	0.92	0.94	0.94

Figure 10 Inter-rater Reliability for Question 13



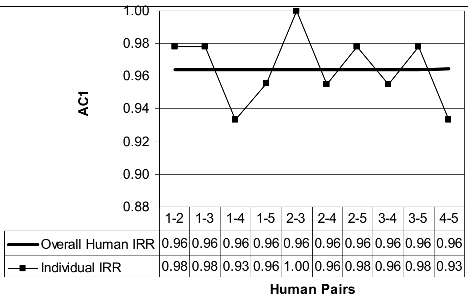
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91	0.91
Individual IRR	0.89	0.95	0.98	0.88	0.89	0.91	0.86	0.98	0.88	0.91

Figure 11 Inter-rater Reliability for Question 14



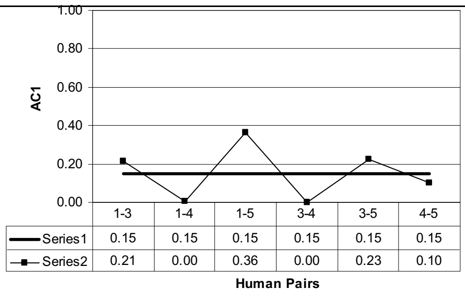
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.40	0.40	0.40	0.40	0.40	0.40	0.40	0.40	0.40	0.40
Individual IRR	0.78	0.33	0.95	0.07	0.31	0.72	0.17	0.38	0.25	0.13

Figure 12 Inter-rater Reliability for Question 15



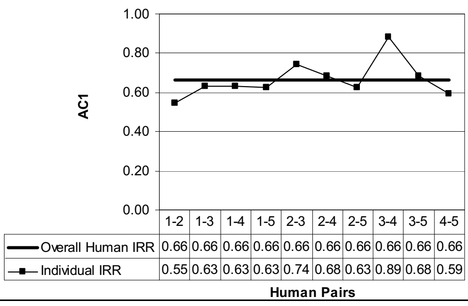
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.96	0.96	0.96	0.96	0.96	0.96	0.96	0.96	0.96	0.96
Individual IRR	0.98	0.98	0.93	0.96	1.00	0.96	0.98	0.96	0.98	0.93

Figure 13 Inter-rater Reliability for Question 16



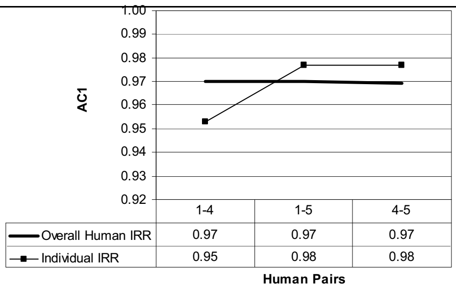
	1-3	1-4	1-5	3-4	3-5	4-5
Series1	0.15	0.15	0.15	0.15	0.15	0.15
Series2	0.21	0.00	0.36	0.00	0.23	0.10

Figure 14 Inter-rater Reliability for Question 17



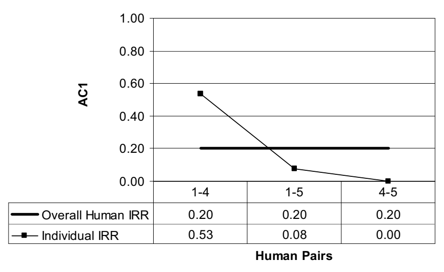
	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Overall Human IRR	0.66	0.66	0.66	0.66	0.66	0.66	0.66	0.66	0.66	0.66
Individual IRR	0.55	0.63	0.63	0.63	0.74	0.68	0.63	0.89	0.68	0.59

Figure 15 Inter-rater Reliability for Question 18



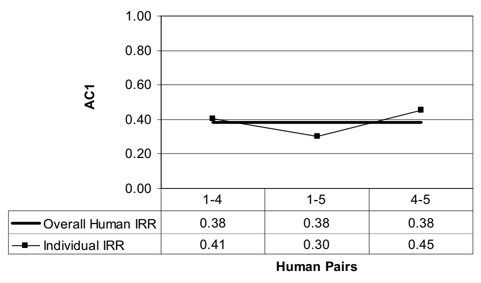
	1-4	1-5	4-5
Overall Human IRR	0.97	0.97	0.97
Individual IRR	0.95	0.98	0.98

Figure 16 Inter-rater Reliability for Question 19



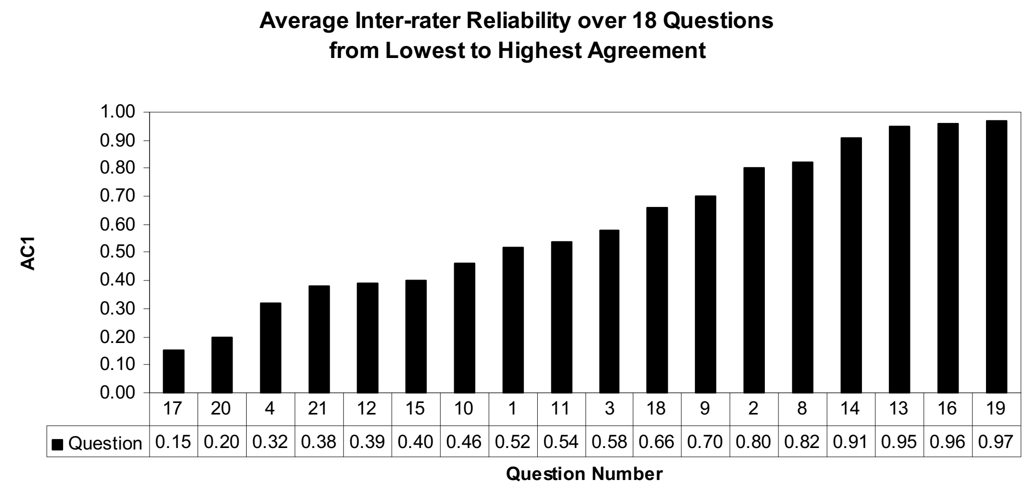
	1-4	1-5	4-5
Overall Human IRR	0.20	0.20	0.20
Individual IRR	0.53	0.08	0.00

Figure 17 Inter-rater Reliability for Question 20



	1-4	1-5	4-5
Overall Human IRR	0.38	0.38	0.38
Individual IRR	0.41	0.30	0.45

Figure 18 Inter-rater Reliability for Question 21



	17	20	4	21	12	15	10	1	11	3	18	9	2	8	14	13	16	19
■ Question	0.15	0.20	0.32	0.38	0.39	0.40	0.46	0.52	0.54	0.58	0.66	0.70	0.80	0.82	0.91	0.95	0.96	0.97

Figure 19 Average Inter-rater Reliability over 18 Questions from Worst to Best

A Focus Group Study of Student Attitudes to Lectures

Michael Hitchens
Department of Computing,
Macquarie University,
NSW 2109, Australia
michaelh@ics.mq.edu.au

Raymond Lister
Faculty of Engineering and Information Technology
University of Technology, Sydney
NSW 2007, Australia
raymond@it.uts.edu.au

Abstract

This paper reports on the findings from focus groups, conducted at Macquarie University, on the attitudes of computing students to lectures. Students felt that two things were vital for a good lecture: (1) that the lecturer goes beyond what is written in the lecture notes; (2) that the lecture is interactive, by which students meant that the lecturer asks if students understand concepts and adjusts the delivery accordingly, and also the lecturer answers the students' questions. The students in the focus groups also discussed what makes for a bad lectures: (1) lecturers reading straight from slides; (2) lecturers who 'blame the students', by saying that students don't work hard enough and are too lazy to turn up to lectures; and (3) lecturers who cover the material too slowly or too quickly. The most prominent reason given for not attending lectures was the timetabling of lectures in such a way that students had too few classes in one day to make the sojourn to university worthwhile. Any university seeking to improve attendance at lectures should perhaps look as much to improving its timetabling practices as it does to improving the practices of its individual lecturers.

Keywords: lectures, student attitudes, focus groups.

1 Introduction

The lecture is possibly both the most iconic element of university education, and the most frequently criticized element of university education. The negative aspects of university lectures are perhaps best summed up in the following well known joke:

The lecture is the process by which the notes of the lecturer are transferred to the notes of the student without passing through the mind of either.

Recognized authorities on university teaching are critical of how lectures are commonly conducted. Biggs (1999, pp. 98–100), citing earlier studies, claimed that attending lectures is a less effective way to acquire information than unsupervised reading, and that lectures are ineffective for stimulating higher order thinking. These problems with lectures, Biggs claimed, are due to the human attention span being limited to 10–15 minutes in passive situations, such as sitting in a lecture theatre. Ramsden (1992, pp. 154–156) made similar claims to Biggs. Ramsden, however, does end his criticism of the lecture with the following caveat:

I would not want to leave anyone with impression that it is impossible to deliver a good lecture, or that I think good teachers should not lecture (though I do think they should do less of it, and for shorter periods).

Perhaps then lectures are not an impossible method for teaching, but a method that few academics currently do well. Certainly, few university academics have received any formal training in how to lecture, and there is a considerable amount that can be taught about lecturing (Bligh, 2000).

In 2006, the Division of Information and Communication Sciences (ICS) at Macquarie University undertook an investigation into student attitudes to aspects of their learning experience. That investigation was wide ranging and included lectures, tutorials, practicals, assessment procedures and feedback. The university's Centre for Professional Development (CPD) was commissioned to facilitate focus groups representative of all four departments in the division, of which computing was one department (the others being Electronics, Physics and Mathematics).

As a result of that study, the Division decided to focus a second study on their students' attitudes to lectures, as the first study pinpointed lectures as a particularly problematic area. Again, CPD was commissioned to facilitate focus groups that were representative of each of the four departments. This paper reports on the outcomes of this second study on lectures.

2 The Focus Groups

2.1 Recruitment and Composition

The focus groups were conducted in August 2007. The students who participated in the focus group sessions were recruited via email and information posted on websites – thus the participants were self-selecting and this should be taken into account when considering the findings. As an incentive to attend, participants were given movie vouchers.

There were eight focus groups sessions in total, of which three groups were comprised of students majoring in computing – each of these three computing groups comprised students from one of the three years of study. This paper describes the findings from the three computing focus groups. All the illustrative quotes below are from the computing focus groups. The composition of the three computing groups is given below.

Copyright © 2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

2.1.1 First Year Computing Group

- 11 participants
- 2 international students
- 0 female students

2.1.2 Second Year Computing Group

- 10 participants
- 5 international students
- 3 female students

2.1.3 Third Year Computing Group

- 7 participants
- 2 international students
- 0 female students

2.2 Focus Group Questions

While the students were encouraged to raise any issues about lectures, several scripted questions were used by the interviewer to initiate and perpetuate discussion:

- What, to you, makes a good lecture? Why?
- What, to you, makes a bad lecture? Why?
- Why do you attend lectures?
- Why don't you, or others, attend lectures? (think about internal and external factors)
- In terms of lecture structure, what would most assist your learning? Why?
- In terms of lecture content, what would most assist your learning? Why?
- What improvements would you make to the current ICS lectures?
- What supporting material (lecture notes, websites, etc) do you currently have access to?
- What supporting material would most assist your learning? Why?

The following sections summarise the discussion that ensued from these questions.

3 "What to You Makes a Good lecture?"

Students nominated several aspects of a good lecture. The most prominent of these aspects are described in the following subsections.

3.1 Adapting to the Students

Students appreciate lecturers who ask if the students understand the concepts and can adapt the lecture delivery according to the feedback the students' provide:

Student: [nominates a particular lecturer] *would refer to something that we had to learn about in layman's terms, in simple terms. So if we didn't understand it then he would explain it even further with different examples.*

<Third Year Group>

Student: *One thing I do like about* [student nominates a lecturer] *is after every section he gives us things to do – like we've got to do it ourselves. So he's not just teaching us – he's making sure we understand while you're doing it.*

<First Year Group>

Student: *I think it's – yeah – good lectures often when they don't have a specific – OK – 'I have to get to slide 38 by the end of the lecture'. They go at the pace of the class.*

Interviewer: *OK – so they've got that flexibility.*

Student: *Yep.*

Interviewer: *So do you think that flexibility comes from good teaching or from knowing the content or just being comfortable.*

…

Student: *I think kind of judging where the students are at by asking them 'do you understand this', you know, 'do you want me to go over anything?'*

Student: *Like the lecturer sort of has to become part of the class almost.*

<Third Year Group>

3.2 Handling Questions

Students consistently nominated the ability to ask questions as a feature of a good lecturer. Among the more experienced students, there was an acknowledgement of the tension between questions and classroom management:

Interviewer: *Is there anything else that would make up a good lecture?*

Student: *More time, more time for questioning.*

Interviewer: *OK.*

Student: *Sometimes I find the lecturers go over the time allocated spot.*

Interviewer: *Trying to get through the content?*

Student: *Because people ask questions in the middle.*

Interviewer: *Oh, OK.*

Student: *And I think that's what most people are there for. If he goes through something and they're unsure they can ask the lecturer.*

Interviewer: *OK. So just a question time to be allocated into the lecture*?

Student: *Yeah.*

<Second Year Group>

Interviewer: *So when you go to a* [computing] *lecture, what makes it work for you? How do you learn the best?*

Student: *I think things like actual student involvement and getting students – getting us to actually think through different examples or things like that – rather than just sitting there for the whole hour...*

Interviewer: *Mm-hm.*

Student: *I think that's a problem though because a lot of the times that – like – the lecturer will say 'OK let's do this example – does anyone have an answer to it?' – Everyone just doesn't say anything anyway. So while that's – like I think it's a good idea to get student* *involvement ...But the thing is that – I think it's – student involvement – yeah – I think is fantastic but when you have a lecture theatre full of – like – a 150 people or something like that it's very hard to, one, hear what the person has to say if they're answering it, but two, actually get people involved as well.*

<Third Year Group>

We note that many lecturers would find an inconsistency between the above student views and their own lecture room experience, where extracting questions from a class can be like extracting teeth.

3.3 Happy, Enthusiastic Lecturers

Students respond positively to lecturers who appear to be happy that they are teaching and are enthusiastic about the material they are presenting.

Student: *... what makes a good lecture is more the lecturer and his attitude towards giving the lecture. ... I've noticed that I've walked out of lectures thinking 'oh that's a good lecture' actually when the lecturer's happy more or less.*

<Second Year Group>

Associated with a positive emotional environment is the notion of humour. Students reported that the occasionally light moment helped refresh their minds:

Student: *... especially when you're doing* [a particular subject] *you are sitting there listening to code and it just doesn't listen to you for the whole hour.*

Student: *Lecturer's ability to bring you out of that for even a second – just say something funny just once in a while.*

Student: *Or even if it's not related just ...*

Interviewer: *Is that like refocussing you almost?*

Student: *Mm. 'Cos they expect us to sit there for an hour and listen to everything and absorb everything- it's kind of hard. But even if it was for like a minute to have a laugh and then you go back to it – it just seems so much better.*

<Second Year Group>

4 "What to you makes up a bad lecture?"

Students nominated several aspects of a bad lecture. The most prominent of these aspects are described in the following subsections.

4.1 Unreadable PowerPoint Slides

Student: *... the graphics of the lecturer's slides. It's – sometimes it's really bad so you can't even see it when you print it out.*

Student: *They can get actually quite complicated sometimes. ... I mean, when you've got code up there fair enough – it's going to be complex. But occasionally you've got something that has like three or four levels of little points and headings and arrows – and it's not very nice to learn from or to read. You're trying to focus on the lecturer but you're also trying to focus and read on the PowerPoint slides. ... Can't do both at the same time.*

<Second Year Group>

4.2 Blame the Students

Students reported feeling alienated by lecturers who make sweeping generalisations about the character of students:

Student: *... I don't turn up to the first week of lectures at all because half of it's ... about how many people failed the last semester, how many people aren't showing up to lectures… they just stereotype the whole group. ... you take it on board quite personally.*

Interviewer: *OK. So they cater towards the people who don't want to learn.*

…

Student: *And obviously the people who have showed up are the ones that do.*

Interviewer: *OK. Well do you guys all go to lectures*?

Student: [several] *Yes.*

<Second Year Group>

4.3 Poor Coordination between Subjects

When lecturers do not have a clear understanding of what the class already know, they may teach too quickly or too slowly:

Student: [Lecturers …] *often they're assuming that – like subjects that we have prerequisites for another subject – they're assuming that because we've done that prerequisite in the past that we know absolutely everything from that subject– there's always going to be things you don't remember perfectly.*

…

Student: *But also on the other hand – sometimes they go over stuff over and over.*

…

Student: *Like we've ... basically conversion methods, like to do with phase conversion and implementation – stuff like that – and all the waterfall methods and all that stuff – I've done that for three years now. Every single lecture it will come up. Its like – OK – I know that – can we move over to something else?*

…

Student: *Yeah ... it just seems like they've got no idea what I've done in the past ...*

Student: *… I find that if I've had the same lecturer through the years like for the subject and then the next one it helps a lot. 'Cos you sort of get to know his style and ... you sort of know what to expect and he knows what he's covered before.*

<Third Year Group>

4.4 The Physical Environment

Students explained that sometimes the lecture room itself was simply unpleasant.

Interviewer: *OK – anything else in your perfect lecture world* ?

Student: *Better seats* [laughter]

Student: [several] *Yes!*

<First Year Group>

Another group cited as a problem the poor audio visual capabilities in some rooms.

5 Why do you attend lectures?

It is perhaps reassuring to hear that some students gave a very positive reason for attending lectures:

> **Student:** *I'm more likely to turn up to the lectures for the subjects that I'm interested in …*
>
> <Third Year Group>

Thus reassured, the remainder of this section describes less positive reasons for attending lectures.

5.1 Assessment Hints

Some students said that they went in case the lecturer spoke about assessment tasks (assignments and exams).

> **Student:** *I feel I'm encouraged to go to lectures when subjects offer something a little more. Like – perhaps – I know this sounds a bit* [inaudible] *but sometimes they go into maybe a little bit of the assessment that's gone and they give you good guidelines. What you don't get from each exercise.*
>
> <Second Year Group>

> **Student:** *I think a lot of people come to lectures to come to hear if they say anything about anything that's really examinable. And like I know if I know that the lecture is going to be about the exam then I'll be a hundred percent. So maybe if you want attendance to go up – say each lecture we're going to be giving you something – like letting you know a little about the assignment that's coming up and how you can approach it and stuff like that – instead of just going on and on and on about theory when clearly we've got an assignment coming up. We're thinking about that – we're not thinking about anything else.*
>
> <Third Year Group>

Sometimes a similar sentiment was expressed more positively:

Student: *One of the good subjects I've had was with the assignment – they didn't just give you everything they're going to give you at the start and then go 'Right. Now work it all out for yourself'. They gave you not much at the start but then progressively they give you six weeks to do it or something. And basically every week or whatever they'd give you more information to help you to do it so you weren't just left doing it on your own and trying to work it out. They'd actually tell you how to do stuff you needed to know for the assignment. ... So you'd start – you'd start your assignment when they first gave it to you and there was only so much you could do because you didn't know it all. But then the next lecture they'd tell you more.*

Student: *That's how you learn ...*

<Third Year Group>

5.2 An Absence of Written Materials

In some cases students said that they had to go to lectures because there was not a set of lecture notes, or a textbook.

6 Why don't people attend lectures?

While students in the focus groups conceded that some students do not attend lectures out of laziness, or because of paid casual work, they described several reasons why even committed students cease attending lectures. In general, many students find that lectures are not a good use of their limited time:

> **Student:** *Well the thing is that a lot of people in third year don't even come to lectures anymore. Do you know what I mean? I'll be honest, I don't come to a lot of the lectures either because I know the lecturer who's taking it – I don't get anything out of it anyway so there's no point. So that's why you get – I mean, first year I went to every single lecture, every single whatever. But then you start to realise ... well I really don't need to be here for this – I'm just going to leave.*
>
> **Student:** *You just get the same amount out of it if you just spend time at home.*
>
> **Student:** *Exactly.*
>
> **Student:** *Than making ...*
>
> **Interviewer:** *Without travel.*
>
> **Student:** *Yep, that's it.*
>
> <Third Year Group>

6.1 No Value Added to PowerPoint Slides

Students want lecturers who go beyond what is written in the lecture notes:

> **Interviewer:** *Do you guys all go to lectures*?
>
> **Student:** *Yep.*
>
> **Student:** *Most of the time* [laughter]
>
> **Student:** *Most of the time.*
>
> **Student:** *Except when my lectures clash.*
>
> …
>
> **Interviewer:** *… do you know people who don't go to lectures and why do you think they don't go*?
>
> **Student:** *Because most of the time they tend to already know what they're up for and what they need to get done.*
>
> **Interviewer:** *And why – how do they know that*?
>
> **Student:** *They read the slides*.
>
> **Student:** *Yeah.*
>
> **Student:** *Yeah – they just assume that everything that's taught is going to be off the slides and just learn the slides and what needs to be done.*
>
> **Interviewer:** *And is that true*?
>
> **Student:** *Most of the time*.
>
> **Student:** *Sometimes – yeah*.
>
> **Student:** *Some of my – well there's one class I haven't been attending ... 'cos I don't feel like I'm learning anything from the lesson*.
>
> <First Year Group>

> **Student:** *I think it's a huge issue. I think it's like – I mean, lectures are supposed to be you listening to the lecturer. I mean, if I wanted to read the slide I'd just read it at home. I wouldn't sit in a lecture room for however many hours to do it. I'm coming to a lecture to hear what they have to say about it.*
>
> <Third Year Group>

Even when the lecturer has an engaging manner, some students will not attend if the lecture if they know the lecturer will only cover the material in the notes:

Student: *... he's actually a good speaker.... But I don't necessarily gain anything more from going to the lecture than just simply reading the textbook or doing the tutorials or things like that. The PowerPoint things that he puts up are fairly good but again, you know, if I read the PowerPoints beforehand I'm OK.*

Interviewer: *There's no value added.*

Student: *Yeah – exactly – there's no value added out of going.*

<Second Year Group>

On reading the above, a lecturer might be tempted to improve their lecturer attendance by not making the lecture slides available, but as the students themselves point out, that may cause its own problems:

Student: *The easy solution for that will be the lecturers will go "OK – well we'll stop putting that content up" but that doesn't solve the problem.*

Student: *Yep.*

Student: *Because* [inaudible]

Student: *Yep, that'll just increase failure rate.*

Student: *Yeah, that'll just increase ...*

Student: *I can guarantee that.*

<Second Year Group>

6.2 Lecturers with Accents

Students were reluctant to attend a lecture if they knew they would struggle to understand the lecturer's accent, but students are not insensitive to the cultural issues of accents:

Student: *I think another important thing is that the lecturer has a clear voice. ... Sometimes if the lecturer is – you can't hear what he's talking about – not interested in the lecture.*

Interviewer: *OK – so when you say clear voice do you mean – just ...*

Student: *The pronunciation.*

Interviewer: *... not accented or just the way they're talking loud?*

Student: *No, I mean the accent.*

Interviewer: *The accent – OK.*

Student: *I know it's something we can't really complain about because it's the lecturer's background.*

Interviewer: *You can complain about it.*

Student: *Yeah – but accent – especially at* [first year] *level – I found really difficult to understand anything. And I found no point in going to lectures as well if I'm going to sit there and struggle to understand the guy when I know most of it is just reading off the lecture slides. I seriously have no motivation to go at all.*

<Second Year Group>

6.3 Class Scheduling

6.3.1 Timing

The times at which classes are scheduled has an impact on student attendance:

Student: *Um – it's not anything really to do with the lecturer – it's sometimes just time. Like – if you have a lecture in the morning and a lecture at night the chances of going in the morning and staying the whole day might be diminished a little bit.*

Interviewer: *OK.*

Student: *Yeah – It's time.*

Interviewer: *Time.*

Student: *Yeah.*

Student: *... Some lecturers – I know it's probably not their fault – but they had one class for that subject and you had to go this, this and this day. And often when you're timetabling you're left with one class in the morning for one hour and the rest of the day off.*

Interviewer: *Mmm.*

Student: *And they're better off thinking, you know, I'll stay at home and read the slides.*

…

Student: *For* [student nominates a particular subject] *I don't know why they separate three one-hour lectures.... in two days. One is eleven o'clock on the Monday and the other one is Tuesday nine o'clock and then one o'clock in the afternoon.*

<Second Year Group>

Student: *It's also – some lectures are on really stupid times. Like, I have a lecture on 6pm to 9 pm on a Friday night. For religious reasons I have dinner at home and all those kind of things on Friday nights. And like – who wants to stay at uni from 6–9 on a Friday night? ... we might be computing students where we like to sit at home on our computers but it still doesn't mean that like …* [laughter]

Student: *Even 9 AM starts four days a week.*

Student: *Yeah there's no way I can get here at 9 at a lecture. 'Cos – I don't know – some people live so far away ... I live in the East which is half an hour drive away. But if I'm leaving – if a lecture is at nine o'clock I'm going to be in traffic for at least an hour.*

Interviewer: *I live in the east. I get here at seven to miss the traffic.*

Student: *Yeah – so that's the thing. So it's just a bit of a problem to have ...*

Student: *And it's worse when you come in at – you make the effort and come in at nine and then, like, he just doesn't do anything and it's just a half an hour lecture and then ...*

Student: *You're like, 'why did I just come?'*

…

Student: [Begins by naming a particular class] *There are only thirteen people in the class and they had a lecture at nine o'clock in the morning and they had to move it because ...*

Student: *… No one would turn up.*

Student: *Three people would turn up* [laughter]. *Because for most people it was the only subject they had the whole day so they didn't bother turning up because it'd take them an hour, an hour and a half to get to uni for an hour.*

…

Student: *I've* [classes] *five days a week but two of them I've only have one hour of class.*

Student: *Last semester I had a 9 o'clock class and then I've got* [no classes] *till like 8pm.*

Student: *I've had classes where I've been here from nine o'clock to nine o'clock.*

Student: *I've had nine to nine.*

Interviewer: *Really?*

Student: *Oh yeah, easy.*

Student: *Yeah.*

Student: *I've had that nearly every year I've had that. This semester I've got that – start at ten.*

Student: *I had – one of those times I had a five hour break in the middle of the day.*

<Third Year Group>

6.3.2 Location

Associated with poor timing of lectures is the poor location of lectures:

Student: *Usually most of our classes and most of our activities are around this building here, and then our lectures are so at the end* [of the campus] *...there are a lot of other classes that people can fit in around here instead of all the way over there.*

Student: *Our classes are not big.*

<Second Year Group>

6.3.3 Length

Part of poor scheduling is also the length of lectures:

Student*: I've got lectures that go for a three hour block. And I think it's just – it's – yeah – it's great to get it all done because you don't have to come in the individual days but on the other hand I would rather coming in different days so you don't have to sit through a lecture for 3 hours. I don't know – it gets a bit annoying when it is for that huge block.*

Interviewer: *Yeah – a lot of people doing one-hour lectures in the younger years were saying 'no, no – we want two-hour blocks'.*

Student*: Yeah – I don't know – by 3rd year I just get a bit annoyed because – I don't know – I lose concentration quite quickly and it's just – it gets to be a lot.*

<Third Year Group>

6.3.4 Relationship with Labs/Tutorials

Poor scheduling and location also adversely affects the relationship between lectures and practical sessions:

Interviewer: *... how do your lectures, tuts and practicals fit in with each other? Do they all ... you know, do you go to lectures and you learn what you need to do for the tuts and pracs and do they fit nicely together?*

Student: *No – it's not really [inaudible]*

Student: *I just went to a prac and we haven't learnt it yet. ... You have a* [subject] *tut and you do the material that you're supposed to learn the next day in that* [subject's] *lecture. So you have no idea what's going on doing* [subject].

Student: *Yeah.*

Student: *Yeah.*

Interviewer: *OK.*

Student: *Sometimes you really don't know. Once you finish your class you rush to the tut or before the tut you go to before the lecture and then you're like in totally two different subjects.*

<Second Year Group>

6.4 Assessment Scheduling: Peak Loads

Sometimes the assessment deadlines stop students from attending lectures:

Student: *… sometimes you get big assignments that are due the same week.*

Student: *You get all your assignments at once.*

Student: *Oh, that's a killer.*

Student: *So sometimes you can't go to lectures because you have to sit in the labs doing the assignments for a week.*

Student: *Every semester I get that ... I got it this semester as well.*

Student: *You sit in the labs the entire week and you never turn up to one lecture the entire week because …*

Interviewer: *You're too busy doing all.*

Student: *You've got all your assignments due at the same time.*

…

Student: *Well, yeah, I understand that we should be doing assignments months and months before but let's be honest like a lot of us don't do that because of whatever.*

Student: *But some of them we don't have the knowledge in the lectures.*

Student: *Yeah, yeah.*

Student: *That as well – we don't have the knowledge for it.*

Student: *We get given the assignment and we don't have the knowledge to start it.*

Student: *We only have the knowledge for it when it's due basically.*

Student: *We'll have maybe two weeks worth to ... get given a month for the assignment but we only learn the knowledge two weeks before it's due.*

<Third Year Group>

7 Recommendations by CPD

As noted earlier, these focus group sessions were conducted by the university's Centre for Professional Development (CPD). As a result of these sessions, the CPD made a number of recommendations to the Division, which are described in the following subsections. These recommendations are taken almost directly from the CPD's report, and were only lightly edited for this paper.

7.1 Professional Development of New Staff

Academics new to teaching, or with less than two years experience in teaching at a tertiary level, undertake some form of professional development in teaching and learning, coordinated by the Centre for Professional Development (CPD).

7.2 Peer Observation

ICS academics take a "professional approach" to teaching, including the incorporation of systematic peer observation of teaching throughout the division.

7.3 Curriculum Review

The division undertake an extensive curriculum review in order to investigate the potential of lectures being more interactive – this could mean more content is available online.

7.4 Recognition of Good Teaching

The Division/Departments recognise and reward good teachers and use them, together with Divisional winners of Macquarie Outstanding Teacher awards as examples of good practice.

7.5 Communication problems

Academics with strong accents/language difficulties are provided with assistance to overcome the problems these pose in the teaching and learning context.

8 Discussion and Further Findings

In this section, we first discuss the CPD recommendations before making some further suggestions of our own.

8.1 Comments on the CPD Findings

With the exception of the recommendation regarding academics with strong accents / language difficulties, the CPD recommendations are not closely related to the evidence in the focus group transcripts. These recommendations are what university Teaching & Learning units across Australia have been advocating for many years.

The CPD recommended professional development of new staff. This is a good recommendation, but we ask '*what in the focus group transcripts suggests that new staff members are a particular problem*?' We find almost nothing in the transcripts concerning the experience levels of lecturers. If anything, what little evidence we did find in the transcripts suggests that it is the older academics, not the new academics that may need the professional development:

Student: ... *don't get me wrong because older people can be really happy and really energetic and really passionate. But, you know, if you're sort of new to something – like when you get a new job you work hard to like, you know, impress and keep it. But I don't really think that – like I think they get older so they just don't care. They just want to hurry up and teach and get out of there.*

<Second Year Group>

8.2 Other Factors in Poor Lecture Attendance

This subsection discusses several factors leading to poor lecture attendance that we found prominent in the transcripts but were not addressed in the recommendations of the CPD.

8.2.1 Timetabling

A prominent reason given for not attending lectures is the timetabling of lectures in such a way that students had too few classes in one day to make the sojourn to university worthwhile. The transcripts also contain student suggestions on how classes could be more attractively timetabled:

Student: ... *you would know that someone who's doing, say, an information systems degree would have to do these subjects. Can't you sort of sync it up so that they're all in the same block? ... Instead of waiting three hours for a lecturer to come and then you don't really need to go to it. Do you know what I mean*?

Student: [The student begins by advocating the adoption within computing subjects of a timetabling structure that the student experienced in a non-computing subject]. … *it's like a two-hour block which is like – the first hour is lectures and stuff like that and afterwards if you've got questions and stuff the lecturer will be there to answer questions – or, say for a computing subject, that he could be in the labs or something like that ...*

Student: *That's like the* [computing subject] *we did it last semester was actually good because you'd have lecture and then for ... the prac … The lecturer and – one of her PhD students I think – they would be in the labs the hour after ... like the whole class would walk to the labs and we'd do like practice stuff together.*

…

Interviewer: *So you had access to them after the lecture.*

Student: *And it was useful because like she was there to help the whole class. Like we could be doing our pracs and go 'OK how do I do this' and she'd be able to help us as well as discuss.*

<Third Year Group>

8.2.2 Coordination of Assessment Load

Students identified heavy peak assessment load as a factor in non-attendance at lectures. One student also pithily suggested a solution:

Student: … *the lecturers need to talk between each other because sometimes you get big assignments that are due the same week.*

Many experienced academics will complain about externally mandated alterations to their assessment schedule, with some justification, as the deadlines for assignments in individual subjects are often bound tightly to the sequencing of the lecture material. We concede that, from the perspective of a single lecturer/subject, coordinating assessment deadlines across subjects might introduce some inefficiency into the teaching of single subject. However, lecturers who are troubled by poor lecture attendance might consider the larger picture; that students attendance at their lectures might improve as a result of coordinating assessment deadlines, and perhaps there is a net gain to be had via the gambit of slightly suboptimal assessment deadlines.

The second author has worked in a faculty (not at his current institution) where assessment deadlines were mandated across subjects, so coordination of assessment deadlines is – not popular, but – possible. If mandated deadlines are too difficult, then at least lecturers could communicate their deadline intentions to each other, and some of them might then adjust their own assessment schedule voluntarily

8.2.3 Coordination of Content Across Years

Students identified the repetition of material in successive subjects as a factor in non-attendance at lectures, and they also discussed the obvious solution:

Student: ... *I don't know if lecturers actually talk to each other to let them know what they've done. ...*

Student: *Yeah – well, I'm sure it happens. But it just seems like they've got no idea what I've done in the past ...*

Coordination across several subjects is difficult, as every lecturer knows that their colleague may have talked about something, but the students may not have learnt it. Never-the-less, there is no harm in academics at least talking to each other to find out what their colleagues claim to have taught.

9 Discussion of Findings

We identified two underlying themes in the focus group transcripts that were not captured within the CPD recommendations:

- Lecturer attitude
- Value-adding

With respect to the first of these underlying themes, we can hardly expect our students to be enthusiastic about attending lectures if we lecturers do not convey a positive attitude about lectures. This might appear self-evident, that a lecturer who is disinterested – or worse negative – about taking the class, is giving a signal to students to adopt the same attitude. From the study results presented in this paper it appears that it is worth reminding ourselves and our colleagues of this self-evident truth.

With respect to the second of the underlying themes (i.e. value-adding), students are increasingly asking questions about what the lecturer experience is providing to them. We should at least be thankful that they are asking questions, if not perhaps the ones we would prefer. Scientific enquiry is not meant to ask the easy questions. The results from this focus group study indicate that students want the lecture to contain material and experiences that could not as easily be gained from private reading of the textbook or notes. The lecturer might achieve such an outcome by many alternatives. Some alternatives are: (1) to make the lecture an opportunity to ask questions of the lecturer; (2) have the students work on problems for which the lecturer then presents a worked solution; (3) provide assessment and examination information. Perhaps the message is that if the lecture would have been given in the same way even if no students were actually present, and especially if the lecturer's attitude was disinterested or worse, then we have a lecture that the students are not keen to attend. Lectures that involve the students, which give them positive reasons to attend, will see their own reward.

10 Conclusion

Computing academics often warn of the dangers of outsourcing software development. Analogous problems arise when academics within a discipline outsource the evaluation of teaching quality to an external unit of their own university. Teaching and learning units have their own perspective – (to use an expression common in teaching and learning units) they see the world through a particular lens – and therefore teaching and learning units bring their own biases to any analysis of teaching within a discipline. We are not advocating that teaching and learning units should be ignored – on the contrary, we strongly advocate that academic disciplines work closely with their teaching and learning units. What we warn against is a complete dependence on an outside source of authority for the evaluation of teaching and learning issues.

Teaching and learning units tend to focus on problems that can be fixed by changes to the practises of the individual lecturer. That focus makes teaching and learning units less aware of the sources of other problems in university teaching, and therefore less likely to identify problems that can only be solved by changes to systemic procedures. For example, in this paper, we identified poor timetabling of lectures and associated classes as a factor in low lecture attendance, but the report from the teaching and learning unit made no recommendations in that regard. On the basis of our reading of the transcripts, we conclude that any university seeking to improve attendance at lectures should look as much to improving its timetabling practices as it does to improving its individual lecturers.

References

Biggs, J. (1999) *Teaching for Quality Learning at University*. Buckingham: Open University Press.

Bligh, D. A. (*2000*). *What's the Use of Lectures*? San Francisco, CA: Jossey-Bass.

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

What Our ICT Graduates Really Need from Us: A Perspective from the Workplace

Tony Koppi[1] Judy Sheard[2] Fazel Naghdy[1] Joe Chicharo[1] Sylvia L. Edwards[3] Wayne Brookes[4] David Wilson[4]

[1]Faculty of Informatics, University of Wollongong, Australia

tkoppi{fazel, chicharo}@uow.edu.au

[2]Faculty of Information Technology, Monash University, Australia

Judy.Sheard@infotech.monash.edu.au

[3]Faculty of Information Technology, Queensland University of Technology, Australia

s.edwards@qut.edu.au

[4]Faculty of Information Technology, University of Technology Sydney, Australia

brookes@it.uts.edu.au, David.Wilson@insearch.edu.au

Abstract

A national Discipline-Based Initiative (DBI) project for Information and Communications Technology (ICT), funded by the Australian Learning and Teaching Council, has sought the opinions of recent graduates of ICT in the workplace to help inform the curriculum. An online survey was devised to question graduates on workplace requirements and university preparation for abilities categorized as: personal/interpersonal; cognitive; business and technical. The graduates in employment have highlighted broad mismatches between the requirements of their professional work in these categories and the preparation for employment they received from university. A regression analysis was used to determine influences on graduates' opinions of the preparation they received at university. The quantitative and qualitative results from this survey could have far-reaching consequences for ICT education and this initiative will enable the development of curricula that ensures graduates are equipped with the skills required by the ICT industry.

Keywords: ICT curriculum, graduate workplace abilities, ICT graduates, professional work requirements, university courses

1 Introduction

This paper reports on a study that is part of the Discipline-Based Initiative (DBI) for Information and Communications Technology (ICT) education in Australia. This national project is based at the University of Wollongong under the directorship of Professors Joe Chicharo (Dean, Faculty of Informatics) and Fazel Naghdy (Head, School of Electrical Computer and Telecommunications Engineering) and is concerned with improving education and the student experience in the broad range of ICT disciplines. The project is partnered by Monash University, Queensland University of Technology, and the University of Technology, Sydney, with the collaboration of the Australian Information Industry Association (AIIA) in parts of the project. It is supported by The Australian Learning and Teaching Council.

The issues and challenges facing the ICT education sector are broad and complex. The context in which these can be explored includes the interrelated areas of high schools, tertiary education providers (which are dominated by universities), industry, professional bodies and government. Furthermore, the discipline area of ICT covers a wide spectrum with engineering-related disciplines at one end and business/commerce-related disciplines at the other.

In spite of the downturn in the early years of this century, the ICT industry has proved to be quite robust and is set to grow in the coming years. There is a renewed optimism for healthy growth of the sector at least during this decade (Newstrom, 2005). The growth is expected to take place concurrently in all four major sectors of ICT i.e., hardware, software, services and communication.

The growth and expansion of ICT so far and its future development have two major impacts on ICT education. The growing ICT sector will require more trained human resources at all levels including maintenance, design, development, implementation and leadership. At the same time, new developments and inventions will create new fields in ICT, which in turn will demand introduction of new courses and training programs at all levels.

The rapid pace of change in the ICT sector has been driving and demanding parallel changes in all facets of ICT education including curriculum, structure, content and delivery. This has been crucial to ensure that courses offered have relevant curricula, address the needs of the ICT industry and produce graduates of immediate benefit in their employment.

Copyright ©2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 95, Margaret Hamilton and Tony Clear Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

This paper is concerned with the ICT curriculum and its relevance to graduates in the workplace. Recent graduates who have been in the workplace from one to five years have been consulted about their curriculum by means of an online survey. A survey of graduates in the workplace was a recommendation from an Australian Universities Teaching Committee (AUTC) project concerned with ICT education (AUTC, 2001). The survey was designed to elicit from graduates in the workplace the abilities they consider as important for successful performance in their current professional work, and to give their perceptions of how well their university course prepared them for these abilities. The results of the survey will enable universities to develop curricula that better prepare their students for employment in the ICT sector. A similar but smaller and narrower study was carried out by Sumner and Yager (2008) in the US on perceived differences between what MIS graduates learned in their degree program and the requirements of their jobs.

The views of these graduates in the workplace represent industry's requirements of university curricula. Any discrepancy between these requirements and the perceptions of university preparation to meet them would reveal a gap between academia and industry. Such a gap between industry and academia was identified by Yen, Chen, Leea and Kohc (2003) and Nagarajan and Edwards (2008). Another perspective of graduate suitability for the workplace is given by employers (such as reported by Hagan, 2004) but that aspect is outside the scope of this paper.

2 Design of the online survey of graduates in the workplace

The online survey of ICT graduates in the workplace was intended to inform universities about the curriculum with respect to industry requirements. The survey questionnaire was developed by the project team. The design was based on that of Scott (2003) and modified for the purposes of this ICT study. The questionnaire was trialled with graduates and academics before release. Notification of the survey was via local Alumni Offices and responses were received from graduates from 15 Australian universities.

The categories under which the survey questions were devised are as follows:

- Personal/interpersonal abilities
- Thinking/cognitive abilities
- Business abilities
- Technical abilities
- Learning and university experience

The essential structure of the survey was three columns on a webpage with statements of abilities relevant to a particular category down the centre, the rating scale of importance of that ability in current professional work on the left, and the extent to which the university course[1] focused on developing that ability on the right (as shown in Table 1). Five point Likert scales were used for the comparison of responses on the left and right of the table. Comparing the left and right sides illustrates how well the curriculum is integrated with the requirements of professional practice. The results tables presented below differ from the online version, which included radio buttons and the high-low order was reversed.

Text entry boxes were provided at the end of each category for respondents to add any comments and other information they considered would be helpful regarding that category.

3 Results and discussion

This paper reports on results from five universities that gave the majority of the total responses. These five universities were from NSW, Victoria and Queensland, and included one Group of Eight (Go8)[2] university and two from the Australian Technology Network (ATN). There were 548 completed responses to the online survey from graduates in the workplace from these five universities. The results are presented in the following tables as percentage values and they are ranked by the high score of the left column. Data were analysed in SPSS and the distribution of responses relating to current professional work was compared to that of university preparation by using the Wilcoxon Test.

3.1 Personal/interpersonal abilities

There were 12 questions in this category. The percentage responses are shown in Table 1. The results are ordered on the left hand column (high responses) for the importance for professional work. For all questions, the graduates gave a higher rating for the importance of the ability for successful performance of professional work than the extent to which the university course focussed on this ability. These differences were significant according to Wilcoxon tests. There were 104 open text responses to this category.

[1] Course in this context means a program of study for a degree or diploma.

[2] The Group of Eight is a coalition of research intensive Australian universities (http://www.go8.edu.au/)

Importance of this for successful performance in my **Current professional work**						Extent to which my **University Course** focused on this ability				
5 high	4	3	2	1 low		5 high	4	3	2	1 low
61	29	10	1	0	Ability to remain calm under pressure or when things go wrong	11	24	30	21	13
55	33	10	2	1	Ability to contribute positively to team-based projects	29	37	24	7	3
51	33	13	3	1	Ability to communicate effectively in writing	23	36	27	11	3
51	31	13	4	1	Ability to speak to groups of people effectively	13	35	28	15	10
49	39	10	2	0	A willingness to face and learn from my errors and listen openly to feedback	19	31	31	12	6
48	28	17	4	3	Ability to work productively with people from a wide range of cultural backgrounds	25	30	27	11	7
46	34	15	4	2	Ability to communicate effectively and appropriately using electronic media	19	37	26	15	4
44	41	12	2	1	A willingness to consider different points of view before coming to a decision	17	36	27	14	5
34	31	25	7	3	Ability to communicate effectively in visual or graphical formats	14	33	31	17	6
32	27	21	10	10	Ability to consider the impact of my actions on the environment	9	16	31	22	23
30	34	20	10	6	Ability to consider the impact of my actions on people in the broader community	9	19	34	22	17
11	14	19	14	43	Ability to communicate in languages other than English	4	6	16	14	59

Table 1. Personal/interpersonal abilities responses given as % and ranked in order of importance (high score) in current professional work

The ability to remain calm under pressure or when things go wrong is clearly seen as the most important ability in the work environment by nearly two thirds of the respondents. None of the open text responses directly commented on this particular ability although there were several comments of a general nature regarding these ‘soft skills’. Some of those comments noted that universities were generally not good at developing these skills (e.g., “University has not focused on interpersonal skills to the extent it should have focused”), and other graduates noted that these were developed in extra-curricular activities within the university, such as in university clubs and societies, and also in part-time jobs and work experience. Participation in extra-curricular activities to develop interpersonal skills has also been advocated by the Engineers for the development of their graduates (King, 2008).

The ability to contribute positively to team-based projects is a high priority in professional work, as indicated by more than half of the respondents. Nagarajan and Edwards (2008) also reported that teamwork is an important requirement at work. A positive comment noting the importance of teamwork and university contribution from a respondent:

> “Team based assignments where individuals were scored on their contribution to the work were very important to ensure one person didn't "carry" the group, but was also not penalised for others short-falls. In IT today, it is rare to be working solo on a piece of work. More and more employers are asking for self-starting team based players, as more companies adopt the Agile project methodology.”

However, in explanation of the significant difference between importance for professional work and university preparation of this ability (Table 1), other comments note that universities were not fully effective in developing teamwork skills for a variety of reasons including differences in ability, experience, attitude and behaviour amongst the students within an environment that is different from that of the workplace.

Written and oral communication skills were also given high importance by more than half of the respondents and were also significantly different from university preparation for these abilities (Table 1). Nagarajan and Edwards (2008) reported that the dominant skill requirement at work was communication (both verbal and written). Many respondents commented on their lack of development of presentation abilities whilst at university.

> "Presentations were usually done in Tutorials - while the first thing I came across (and I told myself I wish we were trained on this ...) was speaking to over 300 people."
>
> "…and only made 2-3 presentations to peers which utilised an over-head projector."
>
> "I also learned the importance of being able to consult effectively through presentation and confident speaking. I am still working on these skills and had I realised the importance in UNI I would have given more of an effort in presentations etc."
>
> "I think more presentation skill should have been taught in university. In the first few subjects there should have been more importance on presentation"

One university was praised and thanked by a respondent for the valuable relevant training received in communication units.

3.2 Thinking/cognitive abilities

Table 2 shows the thinking/cognitive abilities ranked in order of high importance in professional work. There were eight questions in this category. In each case, graduates gave a higher rating for the importance of the ability for successful performance of professional work than the extent to which the university course focussed on this ability. These differences were significant according to Wilcoxon tests.

Importance of this for successful performance in my **current professional work**						Extent to which my **University Course** focused on this ability				
5 high	4	3	2	1 low		5 high	4	3	2	1 low
61	29	8	1	1	Ability to diagnose what is really causing a problem and test this out in action	19	32	30	12	6
55	30	12	2	1	Ability to identify the core issue in any situation from a mass of detail	18	32	31	14	4
52	39	8	1	0	Ability to access and organise information effectively	28	36	27	7	2
49	39	10	2	0	Ability to bring a creative approach to problem solving	19	38	29	11	4
44	37	17	2	1	Ability to keep up to date with relevant developments	17	33	29	15	7
43	35	17	5	1	Ability to represent and interpret information in a variety of formats (e.g., graphical, text or multimedia)	24	35	29	10	2
42	42	13	2	1	Ability to synthesise information into appropriate formats	20	39	32	7	2
34	35	18	9	4	Ability to work equally well in paper-based and electronic-based formats	24	33	30	9	4

Table 2. Thinking/cognitive abilities responses given as % and ranked in order of importance (high score) in current professional work

Out of the 45 written responses, several respondents claimed that these cognitive/thinking abilities are probably the most important part of the university experience and of high relevance to professional work, for example:

> "The things that I use most from my university education are the personal skills and thinking skills. Very little of the content of my degree do I use in my present role."

Almost two thirds of respondents thought that the ability to diagnose what is really causing a problem and test this out in action is of high importance, and more than half of the respondents thought that the ability to identify the core issue in any situation from a mass of detail was also of high importance. These problem solving abilities were generally not considered to have been well developed at university. Some typical responses:

> "Diagnosing problems is a highly developed skill, but was not "taught" at all. We each practiced it on our on solutions, but weren't given the skills."

> "It would be worthwhile in some of the problem-solving subjects, providing opportunities to students to complete the same task using different approaches would prove useful in training creative thinking."

> "I felt somewhat guided at university as to the solution to a problem especially if it was related to a specific topic. Generally I have little or no information and very vague descriptions of the problem."

> "More real world examples of problems would be useful in ICT courses."

3.3 Business abilities

Table 3 shows the business abilities ranked as being of high importance in professional work. There were eight questions in this category. In each case, graduates gave a higher rating for the importance of the ability for successful performance of professional work higher than the extent to which the university course focussed on this ability. These differences were significant according to Wilcoxon tests.

Importance of this for successful performance in my **current professional work**						Extent to which my **University Course** focused on this ability				
5 high	4	3	2	1 low		5 high	4	3	2	1 low
64	25	9	1	1	Ability to understand, appreciate and meet the needs of your clients	16	29	30	16	9
55	34	8	1	2	A willingness to take responsibility for projects including their outcomes	24	37	25	9	6
54	36	8	1	1	Ability to set and justify priorities	20	32	29	14	5
51	33	11	3	2	Knowing how to manage projects into successful implementation	21	35	27	12	5
50	35	11	3	1	Ability to estimate the time required for work-related tasks	20	33	25	15	8
44	35	15	4	3	Having an understanding of how your organisation functions as a business	11	22	31	19	17
39	33	18	5	4	Ability to be flexible and adaptable to frequent changes of employment	11	20	27	22	20
28	33	23	10	6	Ability to translate innovation into a viable business plan	11	22	30	20	17

Table 3. Business abilities responses given as % and ranked in order of importance (high score) in current professional work

Many of the 34 text responses in this category noted the importance of business skills in ICT employment. Some illustrative responses:

> "I score 5 for all of the items because today's competitive is very tight. All are necessary"

> "Business skills are essential to THRIVE (not just survive) in the IT industry. While Uni graduates continue to under-perform in this area, the degrees they showcase will continue to be under-valued and discounted as mere bits of paper."

> "Aside from 3rd and 4th year projects, I found business skills to be lacking from my degree. When I started to manage projects for my company, I found these things difficult and had to learn very quickly from my mistakes."

> "Understanding the business context is essential however I really only did 1 subject that required this but in my programming work that I am doing I MUST speak the same language as the business and demonstrate that I understand their business from their perspective."

However, while acknowledging the importance of business skills, other respondents commented on other important roles of universities.

"However, it's crucial to have a deep theoretical framework to build on, from the outset. And there's not so much luxury to get this in the chaos of working life. That's where time at Uni is such a crucial window of opportunity for learning theory and concepts."

"I did not do a business or management related degree. I did a technical degree where I would not expect these things would need to be covered. I had gained previous business experience which I use for my current job."

"…it would be more beneficial to have a subject give you a grounding in the principles of the different methodologies, rather than trying to teach you to 'be' a PM on day 1."

The challenge for university teachers seems to be one of achieving a balance of basic theory and business skills that will meet the needs of graduates in industry.

3.4 Technical abilities

Table 4 shows the technical abilities ranked as being of high importance in professional work in this category. The extent to which university courses focused on these abilities is significantly different in each case except for the item that is concerned with being able to program in relevant languages.

Importance of this for successful performance in my **current professional work**						Extent to which my **University Course** focused on this ability				
5 high	4	3	2	1 low		5 high	4	3	2	1 low
50	33	12	3	2	Having the technical expertise relevant to my work area	20	32	29	12	8
41	38	16	4	2	Having the practical skills to generate creative solutions to abstract problems	18	32	31	14	6
36	39	18	5	2	Having a critical understanding of theories and principles in a discipline area	26	39	26	8	2
34	32	21	7	5	Having experience with industry-based project work	12	21	27	22	18
28	33	27	8	4	Having numerical skills	19	32	28	13	8
26	28	24	13	9	Having exposure to ICT professionals prior to my current job	10	21	30	22	18
26	25	16	12	21	Being able to program in relevant languages	17	32	28	12	10
23	27	30	11	9	Being familiar with current technologies rather than fundamental theories	10	18	36	23	12
16	22	34	18	10	Having a firm grounding in fundamental theories rather than being familiar with current technologies	18	30	34	12	7

Table 4. Technical abilities responses given as % and ranked in order of importance (high score) in current professional work

Many of the text responses (49 in total) in this category commented that a focus on new technologies and practicalities relevant to the workplace is required. Some typical comments:

"A focus on new technologies available would be good even if it was a brief overview before leaving university to get a job."

"Although the theory of a concept is important, believe that more time / focus should have been provided for the application of the particular theory."

"Most of the course dealt with theory in depth - and failed to provide the practical skills relevant for work."

Other respondents noted the relevance and place of fundamental theories.

"Nevertheless, I feel the theoretical background I got at Uni has put me in a very good position for adopting new technologies."

"University is not TAFE, you should be learning more fundamental theories than current technologies."

"Technical relevancy is well behind in University. Theory is usually good though and that is where I'm ahead of those that did not go to Uni."

Several respondents noted the importance of specific industry skills and qualifications and commented that these should be available to university students.

"Programming is not a large part of my job however scripting is. Advanced scripting in PHP/Bash/Perl would have been useful to me."

"In my 2 years of full-time employment I have been exposed constantly to the .NET platform which was never even considered during my university career."

"During my course, I was introduced to ASP.NET, VB, C# and SQL. While this is good and relevant, it should be noted too that PHP is dominant in the real world."

"It would be hugely useful to introduce students to BOTH ASP.NET and PHP as that would be a lot more relevant to the real world. I'm taking this survey only because I wanted to make this point - its from experience and I know others who share the same sentiments."

"The Technical skills need to be accredited industry skills. Universities need to realise that learning the fundamentals at university is not up to the standard required by the work force."

These diverse opinions of ICT graduates in the workplace emphasises the challenge that university teaching staff have in finding the optimal balance between fundamental theory, practical application, and industry requirements.

3.5 Learning experiences at university

Table 5 shows the learning experiences at university in relation to a set of abilities. The results are ranked as being of high importance in professional work. The extent to which university courses focused on these abilities is significantly different in each case except for the last item, which is concerned with being able to research publications to prepare documents, reports and presentations.

In this category, problem solving abilities rank the highest for professional work, and it appears that being able to solve problems personally is relatively more important than in a group. It is perhaps not surprising that universities match workplace requirements in the ability to research publications since that is a basic academic activity.

Importance of this for successful performance in my **current professional work**						Extent to which my **University Course** focused on this ability				
5 high	4	3	2	1 low		5 high	4	3	2	1 low
54	32	11	2	1	Problem-solving activities on my own	27	39	23	8	3
44	38	13	4	1	Problem-solving activities in a group	25	34	27	9	5
41	30	19	5	5	Working on projects relevant to industry	15	24	27	20	15
40	33	16	7	4	Giving presentations	26	33	25	10	7
36	30	19	7	8	Interviewing clients to ascertain their ICT needs for a project	10	20	25	22	23
33	30	22	10	6	Researching publications to prepare documents/ reports/ presentations	31	35	23	6	5

Table 5. Learning abilities responses given as % and ranked in order of importance (high score) in current professional work

3.6 The university experience

Table 6 shows the responses to a set of statements relating to the university experience. These indicate that the students were generally positive about their university experience. While some text responses stated that university teachers were not always up to date with technological advances, Table 6 shows that a majority of graduates in the workplace (59%) agreed to some extent that the technical content of their degree was current. Some text responses also indicated that university courses did not prepare students well for their work, however almost two thirds of respondents (Table 6) tended to agree that they were well prepared for work. Furthermore, almost two thirds indicated that they consider their ICT qualification has an advantage over qualifications from other disciplines.

Several text responses noted that part-time work (of various kinds) contributed positively to preparation for the workplace. While 24% of graduates apparently did not have part-time work (Table 6), more than half of the remainder indicated that part-time work contributed to their work preparation.

	5 Strongly agree	4 Somewhat agree	3 Neutral	2 Somewhat disagree	1 Strongly disagree	Not applicable
The technical content of my degree was always up to date	22	37	18	14	7	2
My part-time job helped me prepare for the workplace	25	17	17	7	10	24
My ICT qualification has an advantage over qualifications from other disciplines	30	32	19	10	7	4
My university courses prepared me well for my work	27	34	18	12	7	2

Table 6. Responses to aspects of the university experience given as %

The influence of the graduates' ratings of university course focus items (columns on the right in Tables 1-5) on their rating of how well their university course prepared them for their work (fourth item in Table 6 is taken as an indicator of 'satisfaction') was investigated using a stepwise regression analysis. Twenty four of the 43 university course items that produced coefficient values greater than 0.3 when correlated against the preparation for work item were used in the regression. Five of these items produced a model with an R^2 of 0.291, significant at F (5,408) = 33.455 for $p < 0.05$. The regression output is shown in Table 7.

The significant influence of the ability to contribute positively to team-based projects in the workplace with general satisfaction of university courses is supported by many text responses that note the importance of teamwork in professional practice. However, many text responses also criticise how teamwork is managed in universities.

It is noteworthy that two of these items relate to problem-solving activities. This is supported by many text responses, indicating the importance of problem-solving capabilities in the workplace. It seems that university courses which utilise problem solving activities leads to better prepared professionals. This is also recognised for the engineering profession (King, 2008).

Graduates also seem generally satisfied with the technical abilities they developed at university even though they expressed considerable ambivalence towards current technologies and fundamental theories.

Since none of the business items produced a significant influence, it would seem that whatever business skills they learned or didn't learrn at university, they do not contribute to their general satisfaction with their university courses. However, 'working on projects relevant to industry' is significant in this regard, and that would encompass many aspects including practical application, problem-solving, teamwork and business abilities. Because many of the graduates commented on the lack of business abilities they learned at university, this lack of correlation with a measure of satisfaction may simply be taken as support for their comments.

Extent to which my University Course focused on this ability	Standardised Beta	t	Significance (p value)
Ability to contribute positively to team-based projects	0.213	4.573	0.000
Ability to diagnose what is really causing a problem and test this out in action	0.126	2.485	0.013
Having the technical expertise relevant to my work area	0.175	3.568	0.000
Problem-solving activities on my own	0.135	2.729	0.007
Working on projects relevant to industry	0.118	2.339	0.020

Table 7. Regression analysis results relating general satisfaction with university courses (item 4 in Table 6) with university preparation for the workplace

.

4 Conclusion

Our study found that graduates consider a range of abilities from the personal/interpersonal, cognitive, business, technical and learning domains are important for performance of their work. These include communication, teamwork, problem solving, organisation of information, project management, client relationships and technical expertise. However, there were considerable mismatches between what graduates consider to be of high importance for their work and their perceptions of how well universities focused on developing relevant abilities. The free text comments provided explanation of many of these.

While a majority of graduates seem to be satisfied with how their university prepared them for their work, many perceive themselves as being under prepared in terms of personal and interpersonal skills and business abilities. Graduates claimed they were generally well prepared in technical skills but would prefer more exposure to new and emerging technologies.

The perception that graduates are underprepared in communication and other ‘soft’ skills is not necessarily because universities did not provide the opportunities for the development of these skills. As a number of graduates claimed, as students they did not appreciate the importance of these skills for future work and hence did not engage in developing these as fully as they might have.

Future work on these graduate perspectives and those from employers will be used to determine how industry and academia can work together to produce curricula that will prepare graduates for careers in an expanding ICT profession. An outcome of this approach may be more industry involvement in the teaching of undergraduates.

Information from graduates in the workplace indicated that a well-rounded ICT graduate requires relevant technical know-how, workplace experience, problem solving skills and ability to work in a team for success in professional employment. Sumner and Yager (2008) also concluded that students need a balance of technical and non-technical skills for industry relevance. Perhaps the most appropriate final words are from one of the respondents.

> “I really think that universities need to expose their students to the latest technologies as that is the first step in preparing them for full-time employment. The next step is to expose them to a working environment to teach them that excellent grades will only get you so far and that you need to have people skills to help you excel in what you do.”

5 References

AUTC (2001). (Higher Education Division, Department of Education, Training and Youth Affairs) *Teaching ICT, The ICT-Ed Project, The report on learning outcomes and curriculum development in major university disciplines in Information and Communication Technology*, Computing Education Research Group, Faculty of Information Technology, Monash University, pp. 275.

Hagan, D. (2004): Employer Satisfaction with ICT Graduates. *Proc. Sixth Australasian Computing Education Conference (ACE 2004)*, Dunedin, New Zealand, 30:119-123.

King, R. (2008). Addressing the supply and quality of engineering graduates for the new century. Report for the Carrick Institute for Learning and Teaching in Higher Education.

Nagarajan, S. and Edwards, J. (2008): Towards Understanding the Non-technical Work Experiences of Recent Australian Information Technology Graduates. *Proc. Tenth Australasian Computing Education Conference (ACE2008)*, Wollongong, Australia, 103-112.

Newstrom, G. C., (2005). Keynote speech at "Information and Communication Technology (ICT): Prospects for the Future", at the II International Summit on Information and Communication Technologies (ICT): COSTA RICA INSIGHT 2005, cited at http://www.witsa.org/papers/index.htm#5, August 2006.

Scott, G. (2003), Using successful graduates to improve the quality of curriculum and assessment in nurse education, Australasian Nurse Educators Conference, Rotarua, New Zealand, September

Sumner, M. and Yager, S.E. (2008). An Investigation of Preparedness and Importance of MIS Competencies: Research in Progress. *Proc. of the 2008 ACM SIGMIS CPR conference on Computer personnel doctoral consortium and research*, pp 97–100. Available http://portal.acm.org/author_page.cfm?id=81100123452 and viewed 11 September 2008.

Yen, D.C., Chen, H-G., Leea, S. and Kohc, S. (2003). Differences in perception of IS knowledge and skills between academia and industry: findings from Taiwan. International Journal of Information Management 23, 507–522.

Evolution of an International Collaborative Student Project

Cary Laxer
Rose-Hulman Institute of Technology
Terre Haute, IN, USA
cary.laxer@rose-hulman.edu

Mats Daniels
Uppsala University
Uppsala, Sweden
mats.daniels@it.uu.se

Åsa Cajander
Uppsala University
Uppsala, Sweden
asa.cajander@it.uu.se

Michael Wollowski
Rose-Hulman Institute of Technology
Terre Haute, IN, USA
michael.wollowski@rose-hulman.edu

Abstract

International collaborative student projects are inherently difficult for everyone concerned – the students working on the projects, the faculty guiding the students, and the clients submitting the projects. With more and more schools recommending, or even requiring, that their students have some form of international experience in their degree programs, these projects will become more prevalent in helping to educate computing students in the 21st century. Understanding cultural differences between countries helps students have a better appreciation for the global aspects of computing and the issues faced in making software work in an environment they are not used to. This paper discusses the evolution over four years of collaborative projects between computing students at two schools, one in Sweden and one in the United States. The projects are based in courses at both schools that deal with computing in society. We discuss what the faculty teaching the courses and guiding the projects have learned and how they have improved the experience, what the students learn through these projects, and how the clients interact with the students and faculty. Suggestions for further development of these projects are also made.

Keywords: International collaborative projects, computing and society, open-ended group projects, real-world problems

1 Introduction

Computer scientists and software engineers face many challenges with the projects they work on. The problems they solve are mostly open-ended and team-based. More and more frequently the projects are also international in nature, with many multinational corporations using around-the-clock approaches to getting problems solved. The challenge to computer science educators is how to get their students exposure to these types of problems and experience in solving these types of problems, so that when the students complete their degrees they are able to join the professional workforce with the necessary skills to make them useful as quickly as possible to the companies hiring them.

Fuller et al (Fuller, Amillo, Laxer, McCracken, and Mertz 2005) showed that working on international projects facilitated learning for computer science students. The students should understand the needs of the local communities in which their project results will be used. However, in the past there has been little, if any, opportunity for students to experience, and overcome, language and cultural barriers that are inherent in international collaborative projects (Azadegan and Lu 2001).

Likewise, open-ended group projects facilitate student learning by allowing the students to reflect on and apply the fundamental computer science and software engineering principles they have learned to a larger, perhaps less well-defined, problem than they are used to working on (Newman, Daniels, and Faulkner 2003). These projects "can contribute to the development of professional and 'social' skills which will be essential for their future careers" (Newman, Daniels, and Faulkner 2003).

The work reported in this paper combines the desired educational goals of both international collaborative projects and open-ended group projects. The approach strives to overcome the educational shortcomings of international collaborative projects mentioned earlier and tries to provide a richer educational experience for the students.

2 Method

The findings in this paper are based on us working as reflective practitioners. The setting is two non-traditional courses and as such warrants attention in order to better understand the created learning environment. The authors have collected feedback using several mechanisms: examination methods, course evaluations, being part of the project process, and discussions with colleagues. In addition, several faculty colleagues interviewed students to find pedagogical strengths and weaknesses and ways to improve the course over several years. The actions and findings presented in this paper are strongly influenced by the authors' active participation in the computing education research community, as well as a continuing scholarly interest in educational theories. The findings are presented in a narrative form.

3 Background

The Department of Information Technology at Uppsala University has offered a course titled "IT in Society"

Copyright © 2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

(ITiS) since 1998. The overall goal of this course is to provide an understanding of the interactions between technology, users and designers. This involves examining several types of interactions – between users, between designer and user, between user and system, between designer and system – and looking at how other factors influence each interaction. It requires particular attention to interactions between people and to the ways in which technology influences them. For example, we examine interactions between users and how those interactions are affected by the computer systems employed. An important goal of the course is to provide a framework for, and experience in, evaluating social and ethical issues in the use, or construction, of technical products.

The course gives a brief theoretical exposé over areas like organizational psychology and group dynamics, in the context of interactions between users and the role of technology in those interactions. The issues are practiced in corporate projects with real work environments. This course views system design in its social context, showing how technical, psychological and sustainability issues are important considerations in system design, as are health and ethical issues. There is a focus on how to make individual groups of four to seven students function as well as how to structure collaboration between these groups in a wider context. Four years ago this wider context began to include students from the Department of Computer Science and Software Engineering at Rose-Hulman Institute of Technology in the United States, enrolled in a course titled “Computing in a Global Society.”

In this course, the Rose-Hulman students explore the importance and relevance of globalization, especially as it relates to computer science and software engineering. Readings, such as “Globalization and Offshoring of Software” (Aspray, Mayadas, and Vardi, 2006), and seminars provide the mechanisms for exploring this topic. Like the course at Uppsala University, there is a focus on team dynamics and collaboration in a wide context.

An open-ended group project serves as the primary focus of the two courses, around which discussions and lectures are centered. The project is performed for a client external to both departments and has a clear societal impact. For many students in the courses this project is their first exposure to an open-ended assignment of great magnitude. At first this poses a daunting challenge for them. During the first class meeting, when students were asked to introduce themselves and tell what they hoped to learn from the course, most students expressed apprehension about what to expect from the course project. At the end of the course most students had high regard for the experiences they had and what they accomplished, and they clearly indicated they learned a lot from the project and were better equipped to handle such projects in the future. Open-ended group projects enable students to reach higher-order thinking skills (Hauer and Daniels 2008) and it was clear the students in these courses achieved those skills.

3.1 History of the Collaboration

In the fall of 2004 the Associate Dean of the Faculty at Rose-Hulman Institute of Technology was approached by one of the authors (Daniels) at the Department of Information Technology at Uppsala University about the possibility of collaborating on a project that was to investigate the feasibility of designing web-based modules for improving Swedish health care professionals’ abilities to interact with their patients. The associate dean decided this project best belonged in the Department of Computer Science and Software Engineering and asked the department head (author Laxer) if he was willing to pursue it. The two faculty members involved knew each other professionally and decided to attempt the collaboration. Neither department had attempted a collaboration like this on an open-ended project before (although Uppsala University has done international collaborative projects that have been well defined, such as the Runestone project (Daniels, Petre, Almstrum, Asplund, Bjorkman, Erickson, Klein, and Last, 1998, and Last, Daniels, Almstrum, Erickson, and Klein, 2000)).

Students at Uppsala University would receive academic credit for the project through the IT in Society course, in which the project was based. The Rose-Hulman students would earn academic credit through a Special Topics in Computer Science course. Since the nature of this collaboration was unprecedented, student participation at Rose-Hulman was invited – interested students were asked to submit a short application essay about why they were interested, what they hoped to get out of the experience, and what they could contribute. They were made aware that this was a new and experimental course, and that the project was open-ended and not well-defined. Four students were chosen for the initial collaboration.

The project was based at Uppsala University Hospital, thus the main effort would be in Sweden. A United States perspective on health care in the area of patient interaction was desired, which is where the Rose-Hulman students contributed. The research focus was training of health care workers in patient interaction, and the role of technology in patient interaction solutions. The two sets of students collaborated via e-mail and chat programs (such as Skype).

Toward the end of the semester-long project, the four Rose-Hulman students, along with their faculty member and the associate dean, traveled to Sweden for a week of interaction with their Swedish counterparts. The cost of the trip was covered by funds from the Rose-Hulman president, associate dean, and academic department. The students made a presentation on the project to the IEEE education workshop (CeTUSS) at Uppsala University, worked on the project report, and presented the project to the client. Although no software was written or prototype developed, the feasibility study proved successful and the client was very satisfied. The Rose-Hulman students also had a day of cultural activities, allowing them to learn something about the country with which they were collaborating.

For a first try, the collaboration was deemed successful enough that a second collaboration the following fall (2005) would be tried. The project that year consisted of three pilot studies for a European Union-wide project called SPEX (SPreading EXcellence in health care); one of the pilot studies was located in Sweden and based at the Uppsala University Hospital. Academic credit was handled the same way at both institutions as during the first year, and again four Rose-Hulman students were chosen to participate.

The SPEX project looked at ways that medical expertise could be provided by specialists at centers of excellence to general practitioners at points of care, with the goal of reducing the transport costs and time for both the patients and the doctors, as well as getting the needed health care to the patients in a quicker manner. Students were divided into teams, with each team responsible for different facets of the project (technology, information needs, cost, etc.). Each team had a faculty advisor, chosen from among the faculty involved in teaching the courses at each institution.

During that offering of the course students developed some software and used some technology (digital pens and explanograms (Pears and Erickson 2003), developed at Uppsala University) to offer a proposed solution to the project. The project was presented three times – to an IEEE education workshop (CeTUSS) at Uppsala University, to the client, and to a European Union-wide conference on the project in Barcelona. This time two Rose-Hulman students went to Sweden for a week and the other two Rose-Hulman students joined several students from Uppsala University, along with both faculty members, in Barcelona for the final project conference.

Once again the collaboration was considered successful. The faculty committed to continuing the collaboration, and at Rose-Hulman a decision was made to formalize the project into a regularly offered elective course, Computing in a Global Society (which hopefully would attract more students). A decision was also made to have the Rose-Hulman students make two trips to Sweden during the project, one at the beginning of the project and one at the end. Face-to-face meetings for globally distributed teams are advocated in industry to promote collaboration between remote counterparts (Oshri, Kotlarsky, and Willcocks 2008); the faculty thought increasing the face-to-face contact for the students would thus be beneficial. The group of Rose-Hulman students would be split, with some traveling in September and the others traveling in December. Students could make both trips if they agreed to pay the entire cost of the second trip. With the larger enrollment, the department could no longer afford to pay for all expenses for every student, so the students were asked to contribute $500 each toward their trip. Everyone did so.

The September trip proved to be a huge success. The goal was to build team spirit, meet the client, and begin the project planning. The students from both schools bonded well, and by the end of the week all the students felt the students from both schools were committed to the project, which was to improve the quality of electronic patient journals for the Uppsala University Hospital. Studies have shown that improving interpersonal ties improves the collaboration on global projects in industry [(Jarvenpaa and Leidner 1998), (Majchrzak, Rice, King, Malhotra, and Ba 2000), (Robey, Khoo, and Powers 2000)]. The course faculty hoped that the collaboration between students could also be improved, as well as giving the students exposure to this aspect of team dynamics that would make them amenable to it when they began their professional careers. More details on how the team bonding was accomplished are given in section 6.1.

When planning the project students were placed into five teams, with each team having students from both schools on it. A faculty advisor was assigned to each team, who took part in weekly meetings of the team (usually held by Skype). At the end of the project the second trip by the Rose-Hulman students to Sweden took place, and once again, the project was presented to the IEEE education workshop (CeTUSS) and to the client.

During the 2006 visit, the Rose-Hulman faculty member (Laxer) mentioned to the Uppsala University faculty member (Daniels) that he had a sabbatical coming up the following year, and inquired if it would be possible to spend the fall term at Uppsala University, seeing the collaboration from the other side and working to improve the collaboration for both sides. This was well received, and for the fall 2007 the visiting appointment took place. Another colleague at Rose-Hulman (Wollowski) taught the course at Rose-Hulman while Laxer visited Uppsala.

This year the collaboration had a larger number of students (about 40 in total) and undertook two projects, an expansion of the SPEX project to several health areas, and development of a web portal for the rheumatics ward at the Uppsala University Hospital. Students could decide which project they wanted to work on, and each project attracted students from both institutions.

During the semester Laxer interacted with the Uppsala students in class and out. He gave several lectures to the class, and served as a resource about the Rose-Hulman students throughout the term. The constant interaction proved welcome on both sides, and certainly helped to make things run smoothly. Students at Uppsala University were also asked to write weekly reflections (in English) about topics chosen by all the instructors. Asking them to think about different issues associated with cross-cultural collaborations and project work helped them remain focused on the project and the collaboration. The instructors read the reflections and provided feedback on them to the students. It is anticipated that the reflections will be expanded to the Rose-Hulman students next fall.

4 Issues in this Course

There are some key characteristics in the course that we will highlight and discuss. Perhaps the two most prominent aspects of the course are that it is based on the use of open-ended, ill-structured, problems, and that it involves two groups of students from two different kinds of departments.

We will focus on a few particular issues in this section. They are:

- Using real-world collaboration
- Promoting self-confidence
- Having international collaboration
- Avoiding detachment and estrangement
- Focus on process rather than product

Having a real client and a real project from an interesting workplace, e.g. a hospital, is intended to lead to high motivation and maximum exposure to the benefits and challenges of international teamwork. It also gives insights into a potential future profession and provides opportunities to acquire valuable personal skills related to a professional life. It furthermore provides a connection between things learned in the student's education and its use in a real-world setting. Part of dealing with a real-world setting is that clients do not necessarily have a well-formulated problem in mind. Rather, they know that information technology can help them, but they do not know how it can help them. This means that when students ask clients for more detailed specifications, there are no "right" answers and that settings and requirements can change during a project. There is thus no single way to deal with a problem, nor any obvious way to distinguish between what is relevant and what is not. Dealing with such a setting and finding ways to successfully address the problems that arise leads to increased self-confidence about the capability to take responsibility for solving problems in real-life. The increased self-confidence and practice to select and judge potential solutions will ease the transition to future professions.

Part of the complexity in future employment will, for many of our students, be the ability to work in an international setting. Having experienced and mastered problems with working in a multicultural environment spanning several time zones and limited or no opportunities for face-to-face meetings will be essential in such settings. To have overcome the lure to become detached and ineffective due to being overwhelmed by complexity and lack of clear directions is another way to build self-confidence. Having a focus on dealing with the process rather than on the finished product is an important part of dealing with open-ended, ill-structured real-world problems.

These are some of the potentially good outcomes from this course, but it is important to realize that the students are not used to this form of education and that it is important that they get an understanding of these educational goals early on in the course. It is important to realize that part of running such a course is to deal with frustration and uncertainty. The students and teachers should have an understanding of the role of the teacher, and especially the fact that teachers are not there to give answers. There are no easy solutions when key people on the client side of the project are hard to contact and meet with. Misunderstandings due to cultural differences also pose problems that frequently are difficult to solve, nor is there a best way to set up and run a project. There is, however, learning to be gained by reflecting on these issues and the faculty have a role to inspire reflection as well as to give feedback on choices and nudge students when they get stuck.

One of the key challenges of this course is to make each member of the project feel like they are integral members of the team. This is not always easy when using real-world projects as opposed to carefully controlled artificial projects. In this context we wish to discuss the following issues:

- Differences in skill sets
- Language barriers
- Primary location of project

Rose-Hulman students are strong software developers and typically learn by developing software systems, whether through developing specifications or by implementing to a specification. Real-world problems do not always have those components. Some projects are more concerned with usability than with software development. Usability and issues in software adaptation are part of Rose-Hulman's software engineering program and as such those kinds of projects provide a valuable experience for Rose-Hulman students. The students from Uppsala University are typically from the Information Technology program and for them, adaptability and usability studies are much more central to their course of study. As such, for a typical software development project, the skills sets complement each other well. Naturally real-world problems do not always have all of the components of a software project; as such there is a risk of one set of students being not as enamored with the project as the other group.

Some projects require a good amount of interaction with clients and users. If some project members do not speak the language of the users and clients, then other members need to be the eyes and ears for the entire group. This is an interesting and useful experience for all project members and possibly one of the best reasons why students should be encouraged to participate in projects with language barriers.

Each of these issues can be managed, if they occur individually. However, if they co-occur, then there is the risk that their compound effects alienate one of the groups.

Up until now, clients for the course projects have been associated with the university hospital at Uppsala. This means that so far, Rose-Hulman students travelled to Sweden to meet with their fellow students and the clients at Uppsala. While those trips are enjoyed by students from both institutions, it would be desirable to create a situation where students visit each other's countries. The students from Uppsala expressed a desire for such a state of affairs. Additionally, it would ensure parity among the students from both institutions.

This goal is difficult to achieve with real-world problems as the software, at this point, does not have clients in Indiana. In principle, this should not be a problem as one of the primary teaching hospitals in Indiana has a rural health care program for communities within a fairly large reach of Terre Haute. Contacts should be made with the directors of these programs to see about the possibilities of international collaboration. As medical personnel are notoriously busy, this is a formidable challenge. Thought should be given as to project domains which offer both real-world problems and an international scope. With a little bit of luck, standardization and globalization of software development and sales will provide for such opportunities.

5 Pedagogical Development

Each university sets its own requirements for what students can earn academic credit for. Thus, the courses at each school under which the international collaborative project is housed may be different, especially when it comes to lecture material and class discussions. The faculty involved in this project held several discussions about what to cover in class meetings; but, due to the differences in the students and in class scheduling at each institution the two courses were not identical. The following sections describe how the courses have evolved at each school.

5.1 Rose-Hulman Institute of Technology

During the first two years of the collaboration, the number of students allowed to participate in the project was kept small (four per year) to maximize the chance for success. The department had not undertaken an international collaborative project before and had a lot to learn. The course was run as a special topics course, with the students and faculty member handling it more like an independent study and undergraduate research course than a lecture course. Rose-Hulman students discussed in detail with the Uppsala University students what the project requirements were, what they (the Rose-Hulman students) needed to research and report on, and how to get their results back to the students at Uppsala.

It became apparent during those first two years that the client for the projects tended to be the Uppsala University Hospital, and thus the projects had a health-related theme to them. These projects provided opportunities to face and discuss issues that are not usually present in course-based projects, such as information privacy concerns; ethical and legal matters, especially between different countries; and software safety, since harmful consequences to people could occur if the software is not designed properly.

Beginning with the fall 2006 course offering, enrollment was enlarged to twelve students. It was decided to have weekly class meetings in which the discussions mentioned above could take place. The April 2006 issue of *Computer* magazine (*Computer* 2006) focused on engineering clinical software, and provided many good articles to have the students read and discuss. In particular, students were required to read the articles on “Coping with Defective Software in Medical Devices” (Rakitin 2006) and “High-Confidence Medical Device Software and Systems” Lee, Pappas, Cleaveland, Hatcliff, Krogh, Lee, Rubin, and Sha 2006), which were then discussed in class meetings. The project that year dealt with electronic patient journals, and in a very timely way *IEEE Spectrum* published an article on electronic medical records in its October 2006 issue (Charette 2006); this article was posted for students to read as well.

As described earlier, the fall 2006 collaboration was the first one in which the Rose-Hulman students made two trips to Sweden. During the early one, in mid-September, the students were asked to make three presentations to the Uppsala University students and faculty in a combined meeting of the two classes. These presentations focused on (1) electronic patient journals, the focus of the project that term, (2) the Therac-25 incident, to show what could go wrong with medical software development, and (3) Rose-Hulman Institute of Technology, so the Uppsala University students would learn something about the school they were collaborating with and the students that go there. These presentations were very useful, and helped show the Uppsala University students that the Rose-Hulman students were fully committed to the project and the collaboration, and that they would be valuable colleagues in the collaboration. The Uppsala University students also made presentations to the combined class.

For the fall 2007, the course was put in a globalization context. Students were assigned readings from Wikipedia, the book “Global Sociology” by Cohen and Kennedy (Cohen and Kennedy 2000), and the report from the ACM Taskforce on Globalization (Aspray, Mayadas, and Vardi 2006). The book does a wonderful job explaining the long history of globalization and explains the benefits and drawbacks of it. If students see themselves as part of this trend, they have a better understanding of the power and limitations of globalization, ultimately giving them the ability to take advantage of it. Students were again asked to make presentations to the combined class meeting during the mid-September trip to Sweden.

5.2 Uppsala University

There was a change in the organization of the project that occurred before the collaboration with Rose-Hulman Institute of Technology started that is relevant – to have a large overall project with several connections to real-world collaborators. The overall project is broken down into subprojects and if a subgroup runs into problems with getting access to the real-world client they can potentially get a real-world experience through the connections of another subgroup. Another change that also occurred before the collaboration was to use the hospital as the setting for the project. This provided a client that all students readily agreed to spend time on trying to improve the workplace for. The hospital environment also provided an excellent example of where IT is an important part of the workplace and yet needs to be introduced with care.

Even though health care is an endeavor relevant to both Sweden and the United States it is still difficult to find projects where international collaboration can lead to a clear benefit. It is also more difficult to collaborate over the Atlantic Ocean than with local students. We felt that it was important to make both the value of the international collaboration tangible for our students, as well as lower the threshold for active collaboration. Face-to-face meetings were seen as an important ingredient in this effort and funds were allocated to support the stay for the Rose-Hulman students. Regular group meetings were considered important and Skype was introduced as a required media for weekly meetings in 2006. Guest lectures by previous students that could serve as role models for how to communicate in an international setting were also introduced in 2006, and in 2007 this was complemented with a guest lecturer covering cultural aspects of collaboration, especially between Sweden and the United States. Both of these efforts have been seen as valuable and important among the students, as evidenced by the students' written responses to reflective questions posed to them.

Up until now there have been more Swedish students involved in the projects than American students and some subgroups have been purely Swedish. There has been an effort to have the mixed subgroups balanced since we have seen that it is easy for a lone student to become detached from the work of the group. More detail on how we devise group membership is presented in section 6.2.

We have also seen a tendency to view problems as someone else's, that if things do not go well it is due to the other side or someone else in the group. Both the open-endedness and the international component of the course lend themselves to such thinking. Weekly reflections were introduced in 2007 partly in order to address this issue, by making each student reflect on what problems they faced and what they personally could do about it. This had a smaller impact than hoped for, partly due to slow response from the teachers and partly due to some issues not being relevant for all the students. This will, in the 2008 course offering, be addressed by having both general reflections and subgroup specific reflections, with the general reflections being given early and mid-way through the project. The teachers involved in the 2008 course offering will share the load of responding to the reflections and the subgroup specific reflections will be handled in the weekly group meetings instead of as written individual responses.

It is not only collaboration between the students that is difficult; there is also the matter of communication among the faculty involved. Site visits and making efforts to arrange face-to-face meetings (e.g. at conferences) have been arranged. The issues have been both about the collaboration in general as well as about the actual projects.

Another issue that will be addressed in the next instance of the course is the communication with the client. There will be one main contact from the faculty side. Communication between the students and the client will be addressed by agreeing on when the students need consent from the faculty member before agreeing to something from the client.

6 Guidelines for International Student Collaborative Projects

There are several aspects to international student collaborative projects that should be considered to maximize the chances for success and insure the students get a meaningful educational experience. These include how to build team camaraderie and trust, how to manage the project work, and how to handle communications.

6.1 Building Team Camaraderie and Trust

Team projects of all types require interaction among people of varying backgrounds. It is important that team members get to know one another and build a sense of camaraderie and trust amongst themselves. This is easy to do when all team members are in one physical location. When teams comprise students from two locations, in different countries, speaking different languages, it becomes a more challenging task to build that camaraderie and trust amongst the team members. Thus, the faculty involved need to make efforts to facilitate meetings between the students from the schools involved. It is much more meaningful when the students can meet in person, but this necessitates one group of students making an international journey, which has inherent costs – not just financial, but time away from classes as well.

As mentioned earlier, throughout this collaboration the project client has always been based in Uppsala, Sweden, so it has made sense for the Rose-Hulman students to travel to Sweden so they could meet their counterparts as well as the client. The faculty involved in teaching the courses at both institutions, as well as senior academic administrators, all agree that learning something about the country the project is being done in and the culture of the people that live there is an important part of this educational experience. Thus, the visits to Sweden by the American students have always included a day of cultural activities in addition to the socializing with the Swedish students. The social activities included formal and informal meetings in a dedicated laboratory. Students enjoyed exchanging ideas about the project, and were sharing personal experiences. Other social activities contributing to camaraderie included dinners and attending social events in the evening.

The international travel aspect of this course is one of the biggest reasons students are drawn to the course – the Rose-Hulman students eagerly look forward to meeting their Swedish counterparts and working with them. Many of the Rose-Hulman students have not travelled overseas before they took this course; going with their fellow students and faculty member provides an easy introduction to international travel for them. Most of the students have remarked that they could not wait to do further international travel, or to return to Sweden.

During the first year of the collaboration one trip was made to Sweden by the Rose-Hulman students; that occurred at the end of the project, when an IEEE education workshop (CeTUSS) was being held on the Uppsala University campus and the presentation to the client would occur. The faculty valiantly tried to arrange some video conference calls early in the term for the students to meet, but were unsuccessful in doing so. The students introduced themselves via e-mail, and the Angel Learning Management System at Rose-Hulman was used for course collaboration. At the end of the project, while the Rose-Hulman students and faculty were in Sweden, there was a project debriefing amongst the faculty and students from both institutions. Everyone thought the first collaboration was successful. When both sets of students were asked if a trip to Sweden early in the term would have proven useful, both groups of students said they did not think it was necessary. The faculty listened to the students and for the following year kept only one visit, at the end of the project.

At the end of the second year's collaboration, the faculty decided to try two visits to Sweden by the Rose-Hulman students for the following year. The faculty wanted to see if the students getting to know each other early, and working together early in the project would lead to better team dynamics. The early trip occurred in mid-September, about two weeks into the courses at both institutions. Students at Uppsala University arranged some ice-breaker and team-building exercises that were very successful in building camaraderie and trust among the two sets of students. Having the Rose-Hulman students in Sweden early on, meeting with the client and working with the Uppsala University students, also built a better sense of ownership in the collaborative project for both sets of students. Communication throughout the project between the students of both schools was improved from prior years. Both schools' faculty members decided to keep the two visits for the next year's collaboration, and will continue to keep them for subsequent collaborations. The challenge will be to foster and support the interpersonal contacts among the students between the two visits, perhaps by arranging videoconference calls. It is cost prohibitive for educational institutions to have "managerial" (i.e. faculty) visits to the other location to help motivate the team members, as is done in industry (Oshri, Kotlarsky, and Willcocks 2008).

6.2 Project Management and Process

Open-ended group projects, such as those undertaken in these collaborations, are large in scope. For the past two years, there have been between 30 and 40 students total to work on each project (10-12 from Rose-Hulman and 20-30 from Uppsala University). Under the guidance of the course faculty, the students are asked to think about the project and how it could be divided into component parts, with each part having a small team (four to eight students) working on it.

Once the project has been appropriately broken up into smaller parts, students are asked to identify which part they would like to work on. In the past, students were very good at ensuring that each part of the project had a sufficient number of students associated with it. Students were also very good about ensuring that project teams contained students from both schools. This is a good indicator that students enjoy international collaboration. It should be pointed out that due to the nature of how a project is broken up, it may be prudent for students of only one school to work on certain parts, in particular those that require interaction with people who only speak one language (although the collaboration is done in English, and the Uppsala University students speak English very well, many of the Swedish people that get interviewed as part of the research only speak Swedish). The course faculty insure that each project sub-team has adequate and appropriate student representation on it. Each sub-team is asked to choose a leader, who is responsible for making sure that sub-team's work gets done in a timely manner. The team leaders are also responsible for the communication between sub-teams and with the course faculty. Each sub-team is also assigned a faculty advisor.

The sub-teams are asked to meet at least weekly, and to include their faculty advisor in the meetings. Meetings that only involve students and faculty from one school usually take place in person, in either the project room or the faculty member's office. When students from both schools are involved, a computer video chat program, such as Skype, is used to conduct the meetings. People can choose to gather at one location at each school to conduct the meeting, or can choose to have a conference call and be at several locations (this usually depends on the time of day and obligations before or after the meeting).

Each sub-team is responsible for writing their part of the project report (see next section). A report writing team, comprised of a student from each sub-team, brings together the individual parts of the report, adds the appropriate introduction and conclusion sections, and insures that the final report flows smoothly.

6.3 Handling Communications

Communication between team members of any project is vital. When team members are located in different countries that are several time zones apart it takes extra effort to insure communication goes well.

The obvious first choice for communication is e-mail. In addition to the students' regular e-mail accounts, each student had access to the e-mail service included in the Angel Learning Management System at Rose-Hulman. Within this system teams could be formed, which permitted e-mail to be sent to all members of that team. The instructors arranged "Angel teams" that corresponded to each project sub-team, as well as Angel teams for all students, just Rose-Hulman students, just Uppsala University students, all faculty, and everyone. The Angel e-mail system has an option to forward Angel e-mail automatically to the user's internet e-mail address (so students wouldn't need to log in to Angel to see if they had Angel e-mail), and everyone was encouraged to enable this option.

As mentioned earlier, Skype was used to conduct team meetings when team members were located at different locations. With web cameras being inexpensive and Skype supporting video as well as audio, students were encouraged to use web cameras so that they could see each other while they were talking. Some teams experimented with the use of cameras but found it awkward to reposition the camera to the speaker. The camera can instead be used to point to the whiteboard, in case students wish to refer to materials developed there. Students were also encouraged to use Skype between meetings when they needed to talk to one or two other people on their team (or anyone else in the course, for that matter).

Written communication is an important component of the courses. Students are required to produce a project report that detailed the nature of the problem they were solving, what research they undertook to understand the problem and the needs of the client, what their results were, the justifications for the approach they took, and recommendations for further work. As this report is typically very large, each project sub-team was responsible for writing their portion of the report, as well as contributing to the overall structure of the report and to the sections that were not associated with a particular sub-team. Small portions of the report were typically written in Microsoft Word for sharing amongst the sub-team members. Google Docs was used for putting the whole report together and editing by the report writing team. In general, students found Google Docs useful to edit and distribute documents of various natures.

7 Conclusion

Working on open-ended, international collaborative projects is a challenging task for professionals, let alone students, but the ability to do so is important in today's professional workplace. This paper has described the evolution of a collaborative project between students at Uppsala University in Sweden and Rose-Hulman Institute of Technology in the United States. Students have faced and successfully overcome the challenges of working on ill-defined, open-ended projects for real-world clients and of collaborating with their peers in another country and another culture, several time zones away. During debriefings between the students and the faculty at the end of each project the students reported that although they were initially apprehensive about the project and how it would proceed, they were proud of what they accomplished and satisfied with what they learned in the process. The faculty involved agree with this assessment and are continuing to improve the process for future collaborations.

8 References

Aspray, W., Mayadas, F., and Vardi, M (editors) (2006): Globalization and offshoring of software: a report of the ACM job migration task force. Association for Computing Machinery.

Azadegan, S. and Lu, C (2001): An international common project: implementation phase. *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education* (Canterbury, United Kingdom). ITiCSE '01. New York, NY, 125-128, ACM.

Charette, R.N.(2006): Dying for data. *IEEE Spectrum*, **43** (10):22-27.

Cohen, R., and Kennedy, P. (2000): Global sociology. NYU Press.

Computer (2006): **39**(4) IEEE Computer Society.

Daniels, M., Petre, M, Almstrum, V., Asplund, L., Bjorkman, C., Erickson, C., Klein, B., and Last, M. (1998): RUNESTONE, an international student collaboration project. Frontiers in Education Conference, 1998 **2**:727-732.

Fuller, U., Amillo, J., Laxer, C., McCracken, W. M., and Mertz, J. (2005): Facilitating student learning through study abroad and international projects. *SIGCSE Bulletin* **37**(4):139-151.

Hauer, A., and Daniels, M. (2008): A Learning Theory Perspective on Running Open Ended Group Projects (OEGPs). *Proceedings of the Tenth Australasian Computing Conference* (ACE 2008).

Jarvenpaa, S.L. and Leidner, D.E. (1998): Communication and trust in global virtual teams. *Journal of Computer-Mediated Communication* **3**(4) http://jcmc.indiana.edu/. Accessed 22 Aug 2008.

Last, M., Daniels, M., Almstrum, V., Erickson, C., and Klein, B. (2000): An international student/faculty collaboration: the Runestone project. *ACM SIGCSE Bulletin* **32**(3):128-131.

Lee, I., Pappas, G.J., Cleaveland, R., Hatcliff, J., Krogh, B.H., Lee, P., Rubin, H., Sha, L. (2006): High-confidence medical device software and systems. *Computer* **39**(4):33-38.

Majchrzak, A., Rice, R.E., King, N., Malhotra, A. and Ba, S. (2000): Computer-mediated inter-organizational knowledge-sharing: Insights from a virtual team innovating using a collaborative tool. *Information Resources Management Journal* **13(**1):44–54.

Newman, I., Daniels, M., and Faulkner, X. (2003): Open ended group projects a 'tool' for more effective teaching. In *Proceedings of the Fifth Australasian Conference on Computing Education* (ACE 2003):95-103.

Oshri, I., Kotlarsky, J., and Willcocks, L. (2008): Missing links: building critical social ties for global collaborative teamwork. *Communications of the ACM* **51**(4):76-81.

Pears, A. N. and Erickson, C. (2003): Enriching online learning resources with "explanograms". In *Proceedings of the 1st International Symposium on Information and Communication Technologies*, ACM International Conference Proceeding Series, vol. 49, Trinity College Dublin, 261-266.

Rakitin, R. (2006): Coping with defective software in medical devices. *Computer* **39**(4):40-45.

Robey, D., Khoo, H. and Powers, C. (2000): Situated learning in cross-functional virtual teams. IEEE Transactions on Professional Communications **43** (1):51–66.

CeTUSS: http://www.cetuss.se. Accessed 24 Aug 2008.

ITiS: http://www.it.uu.se/edu/course/homepage/ITisam/ht07. Accessed 24 Aug 2008.

A Citation Analysis of the ICER 2005-07 Proceedings

Raymond Lister and Ilona Box
Faculty of Information Technology
University of Technology, Sydney
Broadway, NSW 2007, Australia
+61 (2) 9514 1850
raymond@it.uts.edu.au

Abstract
This paper identifies the most commonly cited conferences, journals and books of the 43 papers within the first three ICER proceedings. A large array of conferences, journals, and books were cited. However, only a small set of journals and conferences were cited frequently, and the majority were only cited within a single paper, which is consistent with a power law distribution, as predicted by Zipf's Law. The most commonly cited books are concerned with education in general (29%) or psychology (20%), while 17% of books are concerned with computer science education and 12% with computing content. The citation results for ICER are contrasted with earlier published citation analyses of SIGCSE 2007 and ACE2005–07.

Keywords: Citation analysis, ICER, SIGCSE, ACE.

1 Introduction

If our human bodies are a reflection of what we eat, then an academic community is a reflection of what its members cite. While there are databases that index the citations of academic publications, such as the Science Citation Index® [Thomson Scientific, 2007], computer science journals and conferences are not comprehensively covered by such databases. Furthermore, such indexes do not tell us what types of conferences and journals are cited by a particular community of researchers, especially a small community like the computing education research community. For example, the established indexes cannot be used to determine whether computing educators cite general educational sources, such as the *Journal of Educational Psychology*, more than they cite non-educational computing journals, such as *IEEE Transactions on Software Engineering*.

In this paper, we investigate which conferences, journals, and books have been most commonly cited in the first three ICER proceedings (i.e. 2005–07).

We, the authors of this paper, have already published two citation analyses of computing education conferences. The first was an analysis of the ACE2005–07 proceedings (Lister and Box, 2008a) and the second was an analysis of the SIGCSE2007 proceedings (Lister and Box, 2008b). One of our findings was that SIGCSE 2007 authors emphasized computing content in their citations rather than educational issues. For example, only 2% of all the books cited were concerned with computer science education and 23% with education in general, whereas 57% of books cited were concerned with computing content. From those statistics, we concluded that:

> *The SIGCSE 2007 citations suggest that the educational epistemology of the SIGCSE community is primarily objectivist, with the focus on course content, rather than a constructivist, student-centred focus on learning.*

We found that the authors of papers in the ACE2005–07 proceedings did not place the same emphasis on content. Just over half of all book citations in those ACE proceedings were to books concerned with general education issues (e.g. the classic texts of Biggs, Ramsden and Bloom). However, there was still an emphasis on computing content, with almost one third of all book citations being to computing texts and reference books.

In this paper, we explore whether those citations patterns in the SIGCSE and ACE proceedings are also present in the papers published in the first three International Computing Education Research Workshops (i.e. ICER 2005–07). Both SIGCSE and ACE are primarily concerned with educational practice, while ICER is a research conference. Thus, a comparison of citation patterns in ICER versus ACE and SIGCSE may shed some light upon the differences, if any, between research and practice in computing education.

1.1 Conference and Journal Rankings

Our interest in the differences between computing education research and practice is not simply intellectual curiosity. For several years, the Australian federal government has been developing a process for reviewing the quality and impact of publicly funded Australian research. The review process is known by the name 'Excellence in Research for Australia', or simply ERA (ARC, 2008). As part of the ERA, the Computing Research and Education Association of Australasia (CORE) has developed a ranking scheme for computing-related conferences and journals (CORE, 2007). All computing journals and conferences in which Australian researchers have published in recent years are to be ranked into a four–tier hierarchy.

Copyright © 2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

The Australian computing academics who have been called upon to make these ranking judgments are not themselves active in computing education research. Consequently, they may not be able to distinguish between education research and education practice, and are therefore likely to make negative judgments about computing education research conferences and journals ("that's a paper about teaching, not research"). By carrying out a citation analysis on ICER papers, and comparing the results to those for the SIGCSE 2007 and ACE 2005–07 proceedings, we hope to be able to articulate a clearer distinction between research and practice in computing education.

1.2 An Overview of ICER Citations

At the time this paper was written, there had only been three ICERs held, in 2005, 2006 and 2007. (The fourth ICER was held in Sydney in September 2008.) In these three ICERs, 43 papers appeared, containing 1130 citations, which is an average of 26 citations per publication. On average, SIGCSE and ACE papers made fewer citations. The SIGCSE 2007 proceedings contained 122 publications with 1398 citations, an average of 11.5 per publication, while the ACE 2005–07 proceedings contained 85 papers with 1475 citations, an average of 17.4 per publication.

Table 1 shows a breakdown of the types of sources in the ICER proceedings and, for comparison, the same figures from the SIGCSE and ACE proceedings. The percentage of citations to conferences and books is about the same for all three conferences. ICER authors cite a higher percentage of journals articles than SIGCSE and ACE authors, while SIGCSE and ACE authors cite a far higher percentage of web pages. Thus ICER authors cite a higher percentage of peer reviewed sources than SIGCSE and ACE authors.

The remainder of this paper focuses on citations to conferences, journals and books.

2 Conference Papers

ICER authors, like SIGCSE and ACE authors, have cited from a diverse array of conferences. In the 43 ICER papers, there are citations to 59 different conferences, which is a ratio of 1.3 conferences to each ICER paper. As shown in Table 2, 56% of those conferences are cited in only one ICER paper. Table 2 also shows that almost 90% of the conferences cited in ICER papers are cited in less than 10% of ICER papers. SIGCSE 2007 papers contain citations to 104 different conferences, which is 0.9 conferences for each SIGCSE paper, but 79% of those conferences were cited in only one paper. ACE has a similar distribution to SIGCSE.

Table 3 shows the percentage of ICER papers citing papers from the following widely known conferences:

- **SIGCSE:** Technical Symposium on Computer Science Education
- **ITiCSE:** Annual Conference on Innovation and Technology in Computer Science Education

Type of Source	ICER	SIGCSE	ACE
Conference	32%	31%	32%
Journal	38%	23%	29%
Book	21%	23%	17%
Web Page	5%	18%	12%
Other	4%	5%	10%

Table 1: The percentage of each type of source cited in ICER 2005-07, SIGCSE 2007 and ACE2005-07.

Percentage of conferences (n) cited in …	ICER (n=59)	SIGCSE (n=104)	ACE (n=121)
only 1 paper	56%	79%	79%
≤ 2 papers	69%	88%	87%
≤ 3 papers	76%	94%	89%
<10% of papers	86%	97%	96%
< 33% of papers	97%	99%	100%

Table 2: Distribution of all conferences (n) cited in each of ICER 2005–07, SIGCSE 07 and ACE2005–07.

Conference	ICER 43 papers	SIGCSE 122 papers	ACE 85 papers
SIGCSE	84%	63%	38%
ITiCSE	65%	20%	20%
ICER	44%†	2%	<1%
ACE	26%	5%	48%
FIE	14%	10%	19%
Koli	12%	Not available	<1%
PPIG	12%	Not available	<1%

Table 3: The percentage of papers in ICER 2005–07, SIGCSE 2007 and ACE2005–07 that cite at least one paper from each of these popular conferences.

† Unlike other percentages in the 'ICER' column of Table 3, this 44% was calculated from the 27 ICER 06 & 07 papers only, since ICER 05 could not possibly cite ICER papers.

- **ACE:** Australasian Conference on Computing Education
- **FIE:** Frontiers in Education
- **Koli:** Koli Calling International Conference on Computing Education Research
- **PPIG:** Psychology of Programming Interest Group (Annual Workshop of)

Conference	Tier	Cites	CitesXSelf	Papers	PapersXSelf
SIGCSE: ACM Special Interest Group on Computer Science Education Conference	A	122	109	36	35
ITiCSE: Annual Conference on Integrating Technology into Computer Science Education	A	58	50	28	23
ICER: International Computing Education Research Workshop	A	28	26	12	12
ACE: Australasian Conference on Computer Science Education	B	15	14	11	10
FIE: Frontiers in Education	B	9	6	6	5
VL: IEEE Symposium on Visual Languages	B	7	6	5	5
CSCL: Computer Supported Collaborative Learning	A+	4	4	4	4
CHI: International Conference on Human Factors in Computing Systems	A+	7	7	4	4
AVI: International Working Conference on Advanced Visual Interfaces	—	5	5	4	4
ESP: Workshop on Empirical Studies of Programmers	—	10	10	4	4
InSITE: Informing Science and IT Education Conference	B	3	3	3	3
OOPSLA: ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications	A+	3	3	3	3
SOFTVIS: ACM Symposium on Software Visualization	—	4	4	3	3
HICSS: Hawaii International Conference on System Sciences	B	4	3	4	3
VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing	A	7	6	4	3
ICSE: International Conference on Software Engineering	A+	3	3	3	3
Koli: Koli Calling	B	5	2	5	2
PPIG: Psychology of Programming Interest Group (Annual Workshop of)	B	6	2	5	2
ICLS: International Conference of the Learning Sciences	—	2	2	2	2
ICFP: International Conference on Functional Programming, ACM SIGPLAN	A+	2	2	2	2
International Seminar on Software Visualization	—	3	3	2	2

Table 4: All conferences cited by more than one paper (excluding self-citations) in the ICER 2005, 2006 and 2007 proceedings. The columns show the CORE tier (a dash appears where CORE have not assigned a tier), total number of citations to the conference ("Cites"), total number of citations to the conference, excluding self-citations ("CitesXSelf"), total number of papers that cited that conference ("Papers"), and total number of papers that cited that conference, excluding self-citations ("PapersXSelf"). The list is ordered (descending) on the last column.

Conference	Average	ICER	SIGCSE	ACE
SIGCSE	90	0.4	0.8	0.3
ITiCSE	53	0.5	0.5	0.5
ACE	38	0.3	0.2	0.8

Table 5: Normalized citation counts in ICER, SIGCSE and ACE (columns) of SIGCSE, ITiCSE and ACE (rows).

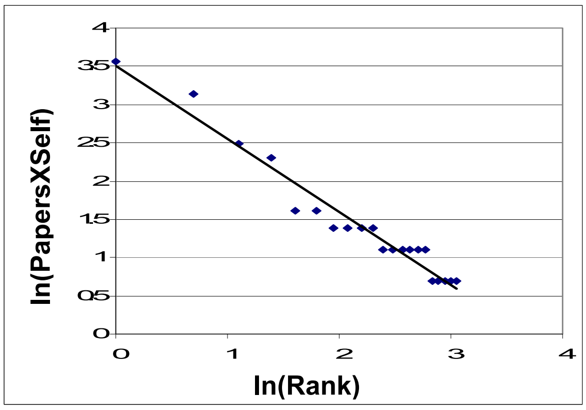


Figure 1: A plot of the logarithm of PapersXSelf vs. the logarithm of the rank of the 21 conferences from Table 4.

Table 3 shows that SIGCSE was the most cited conference by both ICER and SIGSCE 2007 authors. For ACE authors, ACE itself was the most popular conference to cite, but when self-citations are ignored, the citation rate by ACE authors to ACE papers drops from the 48% to 32%, and thus when self-citations are ignored SIGCSE is also the most popular conference to cite for ACE authors.

Among SIGCSE 2007 papers, conference citations to SIGCSE papers are not simply the most frequent—SIGCSE citations are dominant, with the percentage of SIGCSE citations being 3 times higher than the next most popular conference. While ICER authors cited SIGCSE papers even more often than SIGCSE 2007 authors, ICER authors also cited several other conferences extensively.

When considering citation data to determine the popularity of conferences, allowance should be made for two possible sources of distortion, especially for a small, young conference like ICER. Self citation is one source of possible distortion. (The data presented in Tables 1, 2 and 3 includes self citations.) Another source of distortion is the possibility that a conference may be cited in only one paper (or a very small number of papers) but that paper cites several papers from that same conference. Table 4 presents alternate forms of citation data, showing the effect of these forms of distortion on ICER data. With regard to the first source of distortion, self citation rates are not high. With regard to the second source of distortion, it appears that, when an ICER author cites a paper from a popular conference series, they often cite other papers from that same conference series. For example, among the 36 ICER papers that cite a least one SIGCSE, there is an average of 3.4 citations to SIGCSE papers.

When considering citation data to determine the popularity of conferences, allowance should also be made for the differing sizes of conferences—a large conference might have more papers cited than a small conference simply because the larger conference has more papers. In our earlier paper on the citation analysis of SIGCSE 2007, we calculated the average number of papers per year for SIGCSE, ITiCSE, and ACE, in the three years 2003─05, which were 90, 53 and 38 papers respectively. (In more recent years, the typical number of papers in ACE has fallen, but that does not significantly affect our citation analysis here). In this paper, we use those average yearly figures to calculate normalized citation data for SIGCSE, ITiCSE, and ACE. For example, Table 3 shows that 84% of the 43 ICER papers (i.e. 36 ICER papers) cited at least one SIGCSE paper. Those 36 papers are 0.4 of the average number of papers per year for SIGCSE (i.e. 0.4 of 90). Table 5 presents all the data normalized in this way. With this correction made for the size of the conferences, it can be seen that ITiCSE is a more popular source of citations for ICER authors than SIGCSE, and even ACE is three quarters as popular as SIGCSE. Also, ITiCSE is more popular than SIGCSE as a source of citations for ACE authors.

Figure 1 is a log-log plot (to base e) of the PapersXSelf column of Table 4 versus the rank of the 21 conferences from Table 4 (i.e. ranked on PapersXSelf). The plotted points are a good fit to a regression line, which suggests that the distribution of the number of ICER papers citing a particular conference is broadly consistent with the well known power law distribution for citations (Redner, 1998; Tsallis & de Albuquerque, 2000). Such power law distributions are often referred to as Zipf's Law. In Figure 1, the slope of the line of best fit is approximately -1.

2.1 Discussion of Conference Paper Results

In our paper on the SIGCSE 2007 citation analysis, as a consequence of finding this great diversity of citation sources, and also as consequence of the influence upon us of Becher and Trowler (2001), we made the following conclusion :

> *... computer science education (at least how it is practiced by SIGCSE 2007 authors) is a less highly structured, less specialized and slower moving sub-discipline than other aspects of computing.*

We now retract that conclusion, or at least we retract that the diversity of citations in SIGCSE 2007 is evidence that computer science education is a less highly structured, less specialized and slower moving than other aspects of computing. Since writing the above conclusion, we have carried out similar citation analyses for three other conferences, all non-education conferences that are part of the Australasian Computer Science Week (ACSC, ADC and AUIC). For details of the analysis of those other three conferences, see the papers appearing in those respective conference proceedings (Lister & Box, 2009a, 2009b, and 2009c). For each of the six conferences we have analysed—

SIGCSE, ACE, ICER, ACSC, ADC and AUIC—the number of papers citing a particular conference is broadly consistent with a power law distribution. Such a distribution is known to be a property of many conferences, across many disciplines.

Table 4 demonstrates a positive relationship between the CORE conference rankings and the citation rates to conferences from the ICER 2005–07 papers. For example, the three most cited conferences (SIGCSE, ITiCSE and ICER) are all ranked "A" by CORE, which is the second highest category of the five conference rankings (A+, A, B, C and L(ocal)).

3 Journal Papers

As was the case with citations to conferences, ICER, ACE and SIGCSE authors cite from a diverse array of journals. In the 43 ICER papers, there are citations to 132 different journals, but Table 6 shows that just over half of those journals (56%) received exactly one citation in ICER papers, and 90% of journals were cited in less than 10% of the ICER papers. SIGCSE and ACE citations exhibit a similar distribution.

Not only do ICER authors cite more journal papers than SIGSCE 2007 authors (1.7 times as many; see Table 1), but ICER authors also cite the popular journals more often than SIGSCE 2007 authors cite those same journals. This is illustrated in Table 7, which shows that no single journal is cited in more than 20% of SIGCSE 2007 papers, whereas one journal (SIGCSE Bulletin) is cited in more than half of ICER papers and three journals are cited in more than a third of ICER papers. ACE authors also cite SIGCSE Bulletin much more often than SIGCSE authors.

Table 8 shows more comprehensive information about a larger list of journals cited in ICER papers. That table provides for an assessment of the degree of possible distortions due to self-citation, or to multiple citations of the same journal in one paper. Neither form of distortion has a marked effect on the analysis below.

3.1 The SIGCSE Bulletin (Non-Conference)

For ICER, SIGSCE and ACE, the most popular journal is the SIGCE Bulletin. (In this subsection, we ignore the Journal of Computing Science in Colleges, which is cited in 20% of SIGCSE 2007 papers, for reasons discussed below in the subsection devoted to that quasi-journal.).

The SIGCSE Bulletin appears four times a year, but two of those issues are the "conference issues", the SIGCSE and ITiCSE conference proceedings. The results in Tables 6, 7, 8 and 9, are calculated from the two "non-conference issues" of the SIGCSE Bulletin.

Only 17% of SIGCSE 2007 papers cited a paper from the non-conference issues of the SIGCSE Bulletin, which is close to the 20% figure (from Table 3) of SIGCSE 2007 papers that cite the ITiCSE conference proceedings. Both of those percentages are far below the 63% (from Table 3) of SIGCSE 2007 papers that cite papers from earlier SIGCSE conferences. This is surprising, given that all SIGCSE members receive each year all four issues of SIGCSE Bulletin. Either many SIGCSE 2007 authors are not SIGSCE members (which seems unlikely) or they are regular SIGCSE conference attendees who pay more attention to the papers they hear at the conference than the papers that arrive in the post. In the first instance, it is only human to pay greater attention to what we hear than what arrives in our over-flowing post boxes (and which may never be opened). However, as part of writing a paper, one would have expected a SIGCSE 2007 author to perform at least a small literature search, and the SIGCSE Bulletin issues that arrive in the post would be an easy and logical place to start.

Percentage Cited In …	ICER (n=132)	SIGCSE (n=135)	ACE (n = 190)
Only 1 paper	56%	77%	69%
≤ 2 papers	76%	89%	87%
≤ 3 papers	86%	93%	92%
< 10% of papers	90%	98%	99%
< 33% of papers	98%	100%	100%

Table 6: Distribution of all journals (n) cited in each of ICER 2005–07, SIGCSE 07 and ACE2005–07.

Journal	ICER 43 papers	SIGCSE 122 papers	ACE 85 papers
SIGCSE Bull.	63%	17%	34%
CACM	40%	17%	12%
Comp. Sci. Education	37%	10%	14%
J.Educ. Psychology	16%	3%	—
Comp. Res. News	9%	8%	—
IEEE Computer	7%	6%	7%
JERIC	7%	5%	—
J. Comp. Sci. in Colleges	5%	20%	11%

Table 7: The percentage of papers in ICER 05–07, SIGCSE 07 & ACE 2005–07 that cite at least one paper from each of the popular journals

Table 7 (when compared to the ICER data in Table 3) shows that ICER authors, like SIGCSE 2007 authors, have a preference for the SIGCSE conference proceedings, but not to the same degree as SIGCSE 2007 authors. Table 3 shows that 84% of ICER papers cite a paper from SIGCSE conference proceedings, and Table 7 shows that 63% of ICER papers cite a paper from the non-conference issues of the SIGCSE Bulletin—a difference of approximately 20% for ICER papers, compared to a difference of almost 50% for SIGCSE 2007 papers. Also, this 63% is very close to the percentage of ICER papers that cite the ITiCSE conference (65%, from Table 3), which might indicate that ICER authors do read the issues of the SIGCSE Bulletin that arrive in their post box.

Journal	Tier	Author	Cites	CitesXSelf	Papers	PapersXSelf
SIGCSE Bulletin	C	✓✓	59	53	27	25
Communications of the ACM	B		31	31	17	17
Computer Science Education	A	✓✓	26	21	16	14
Journal of Educational Psychology	—		16	16	7	7
ACM Computing Surveys	A*		6	6	6	6
Journal of Visual Languages and Computing	A		16	13	7	6
Journal of Computer Science Education	—	✓	6	5	5	5
Journal of Educational Computing Research	C	✓	10	9	5	5
International Journal of Human-Computer Studies	A		5	5	5	5
Computers and Education	A	✓	9	7	6	5
Cognitive Science	—		10	10	5	5
Educational Psychologist	—		8	8	5	5
J. Computing in Small Colleges / J. Computing Science in Colleges	—	✓✓	4	4	4	4
J. Experimental Psychology: Learning, Memory, and Cognition	—		4	4	4	4
Journal of the Learning Sciences, The	—		4	4	4	4
Expert Systems	C		10	9	4	4
Informatics in Education, An International Journal	C	✓✓	5	4	4	4
IEEE Transactions on Education	B	✓✓	5	4	5	4
Computing Research News	—		5	5	4	4
American Psychologist	—		3	3	3	3
IEEE Computer	B		3	3	3	3
Journal of Computers in Maths and Science Teaching	—	✓	3	3	3	3
Psychological Review	—		3	3	3	3
Review of Educational Research	—		3	3	3	3
Studies in Higher Education	—		3	3	3	3
Contemporary Educational Psychology	—		5	5	3	3
Human-Computer Interaction	A*		3	3	3	3
IEEE Transactions on Software Engineering	A*		3	3	3	3
Cognitive Psychology	—		3	3	3	3
Computers in Human Behaviour	—		3	3	3	3

Table 8: All journals cited by three or more papers (excluding self-citations) in the ICER 2005, 2006 and 2007 proceedings. The columns show the CORE tier ("—" where CORE have not assigned a tier), (column "Author" is explained in the text), total number of citations to the journal ("Cites"), total number of citations to the journal, excluding self-citations ("CitesXSelf"), total number of papers that cited that journal (column "Papers"), and total number of papers that cited that journal, excluding self-citations ("PapersXSelf"). The list is ordered (descending) on the last column.

Journal	Avg	ICER	SIGCSE	ACE
Computer Science Education	15	1.1	0.8	0.8
SIGCSE Bulletin (refereed)	43	0.6	0.5	0.7
SIGCSE Bulletin (all)	69	0.4	0.3	0.4
J. of Comp. Sci. in Colleges	272	0.01	0.1	0.0

Table 9: Normalized citation counts in ICER, SIGCSE and ACE of three computing education journals.

3.2 Computer Science Education

Table 7 shows that the second most cited specialist computer science education journal is Computer Science Education, by ICER, SIGCSE and ACE authors. (CACM is not a specialist computer education journal, and it is discussed separately, below). While 37% of ICER papers cited a paper from Computer Science Education, only 10% of SIGCSE 2007 papers and 14% of ACE papers did the same.

Of course — as was also the case with conference citations—a large journal might have more papers cited than a small journal simply because the larger journal has more papers. In our earlier paper on the citation analysis of SIGCSE 2007, we calculated the average number of papers per year, over the three years 2003─05, for the three journals listed in Table 9. For example, Table 9 shows that Computer Science Education published an average of 15 papers a year over 2003–05. In this paper, we use those average yearly figures to calculate normalized citation data for the three journals in Table 9. We calculated two averages for SIGCSE Bulletin. One of the averages is for all papers published (see "SIGCSE Bulletin (all)" in Table 9). The other average excludes articles like the invited columns and is just for the refereed papers, including the working group papers (see "SIGCSE Bulletin (refereed)" in Table 9). With the citation data thus normalized for the size of each journal, it is clear from Table 9 that Computer Science Education is the most popular source of citations for ICER, SIGCSE and ACE authors.

3.3 Journal of Computing Sciences in Colleges

The Journal of Computing Science in Colleges began with a different name—Journal of Computing in Small Colleges. In this analysis, we use its current name to refer to papers published under either name.

Despite its name commencing with the word "Journal", the Journal of Computing Sciences in Colleges is really an aggregated set of conference proceedings. It contains the proceedings for each of the ten regional journals sponsored by the Consortium for Computing Sciences in Colleges. It is therefore not clear whether the analysis of citations to it should be included in the journal analysis, or should instead be included in the conference analysis. We have chosen arbitrarily to include it as part of the journal analysis.

In terms of the absolute number of citations, the Journal of Computer Science in Colleges is the most cited journal in SIGCSE 2007 papers (20%, see Table 7), but it also publishes far more papers than the other journals, and when citation rates are normalized, this journal ranks lowest among the journals listed in Table 9. Even when normalized, the Journal of Computer Science in Colleges remains a significant source of citations in SIGSCE 2007 papers, but it barely registers as a source of citations for ICER authors.

3.4 Communications of the ACM

We were surprised by the prevalence of citations to CACM articles in the ICER papers, just as we were surprised by its prevalence in our earlier analyses of SIGCSE and ACE. Since then, we have found that CACM is also highly cited in ACSC, ADC and AUIC (Lister and Box, 2009a, 2009b, and 2009c).

The complete set of 23 CACM articles cited in the ICER papers is:

- **Brooks** (April 1980) *Studying programmer behavior experimentally: the problems of proper methodology*
- **Bayman & Mayer** (September 1983) *A diagnosis of beginning programmers' misconceptions of BASIC programming statements.*
- **Camp** (October 1997) *The incredible shrinking pipeline.*
- **Campbell & McCabe** (March 1985) *Predicting the success of freshmen in a computer science major.*
- **Denning** (December 1989) *A debate on teaching computing science.*
- **Denning** (August 1981) *Smart editors.*
- **Denning** (November 2003) *Great principles of computing.*
- **Denning & McGettrick**. (November 2005) *Recentering computer science.*
- **Dijkstra** (December 1989) *On the cruelty of really teaching computing science.* Cited twice.
- **Goldberg et al**. (December 1992) *Using collaborative filtering to weave an information tapestry.*
- **Evans & Simkin** (November 1989) *What best predicts computer proficiency?* Cited twice.
- **Guzdial & Soloway** (April 2002) *Teaching the Nintendo generation to program.* Cited four times.
- **Hu** (February 2005) *Dataless objects considered harmful.*
- **Kramer** (April 2007) *Is abstraction the key to computing?*
- **Mayer** (November 1979) *A psychology of learning BASIC.*
- **McDowell et al.** (August 2006) *Pair programming improves student retention, confidence, and program quality.* Cited twice.
- **Moulton & Muller** (January 1967) *DITRAN—a compiler emphasizing diagnostics.*
- **Shantz et al.** (January 1967) *WATFOR—The University of Waterloo FORTRAN IV compiler.*

- **Shneiderman et al.** (June 1977) *Experimental investigations of the utility of detailed flowcharts in programming.*
- **Soloway** (September 1986) *Learning to program = learning to construct mechanisms and explanation.* Cited twice.
- **Soloway, Bonar & Ehrlich** (November 1983) *Cognitive strategies and looping constructs: an empirical study.* Cited twice.
- **Teitelbaum & Reps** (September 1981) *The Cornell program synthesizer: a syntax-directed programming environment.*
- **Westfall** (October 2001) *Hello, world considered harmful.*

Of these 23 CACM articles, we regard 10 of them as a being education research papers, either reporting an original research result, or reviewing the outcome of research. Another 9 of these CACM articles are not research papers, but are opinion pieces, often written by prominent members of the computing education community (with several of these articles articulating quite sophisticated pedagogical opinions). The remaining 4 of the above 23 CACM articles are technical perspectives, usually about a piece of software that may be helpful for teaching.

Many of the above 23 CACM articles are old. Half of the articles are from 1989 or earlier, with four published before 1980. Only 2 of these articles were published in the 1990s, and 7 were published in this millennium.

3.5 Discussion of Journal Paper Results

A sharp difference between ICER and SIGCSE 2007 citation patterns is the frequency of citations to journals. Only 17% of SIGCSE 2007 papers cite the two most popular computing education journals, SIGCSE Bulletin or Computer Science Education.

Figure 2 is a log-log plot (to base e) of the PapersXSelf column of Table 8 versus the rank of the 30 journals from Table 8 (i.e. ranked on PapersXSelf). The plotted points are a good fit to a regression line, which suggests that—like the earlier plot for conferences—the distribution of the number of ICER papers citing a particular journal is broadly consistent with a power law distribution. In Figure 2, the slope of the line of best fit is approximately -0.7.

3.6 Publishing and the CORE Rankings

In Table 8, the column headed "Author" indicates the suitability of each of these journals for a computing education researcher seeking to publish a paper. Two ticks indicate that the journal is highly suited to a paper on computing education research. One tick indicates that a computing education research paper could appear in that journal, but the journal is more focussed upon the use of computers in education, possibly in any discipline, and not with the teaching of computing.

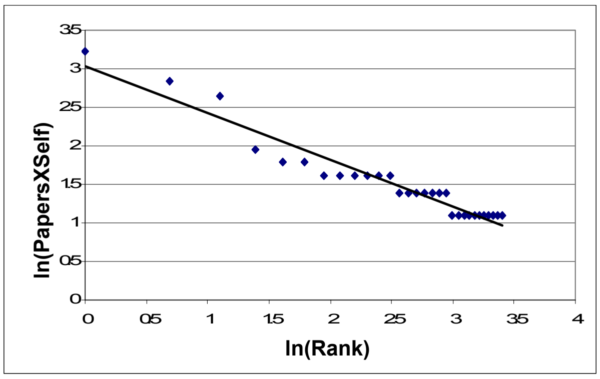


Figure 2: A plot of the logarithm of PapersXSelf vs. the logarithm of the rank of the 25 journals from Table 8.

The journal 'Computer Science Education' is both highly suited and is ranked as a tier 'A' journal by CORE. It is therefore the journal in which most Australian computer education researchers will aspire to publish. However, as this journal only publishes around 15 papers each year, it will also be a very hard place to publish, and Australian computer education researchers will need to look for other journals.

The 'Journal of Computer Science Education' is not the same as the journal discussed above. According to Ulrich's Periodical Directory (Ulrich, 2008) papers in this journal are aimed at those teaching computer science at the pre- college level.

Other computing education journals not listed in Table 8 include:

- Journal of Information Technology Education, ranked 'B'
- ACM Journal on Educational Resources in Computing, ranked 'C'.
- International Journal of Information and Communication Technology Education, ranked 'C'
- International Journal of Information Technology Education, ranked 'C'
- Journal of Informatics Education and Research, ranked 'C'
- Journal of Information Systems Education, ranked 'C'.

Type of Book	ICER	SIGCSE	ACE
Education	28%	23%	51%
Psychology	20%	6%	6%
Computing Content	17%	57%	29%
CS Education	17%	2%	3%
Research Methods	9%	3%	4%
Social	3%	5%	7%
Other	7%	9%	1%

Table 10: The frequency distribution of different types of books cited in the ICER and SIGCSE 2007 proceedings.

Times Book Cited	No. of Books		Cum. %age
1	143		82%
2	20		93%
3	6		97%
4 (10% of the 43 ICER papers)	2	Bransford, Brown, and Cocking (2000) *How People Learn* [education] Hoc, Green, Samurcay & Gilmore (1990) *Psychology of Programming* [CS education]	98%
5	1	Fincher and Petre (2004) *Computer Science Education Research* [CS education]	98%
6	1	Margolis, J. and Fisher, A. (2002). *Unlocking the Clubhouse* [CS education].	98%
8	1	Soloway and Spohrer (1989) *Studying the Novice Programmer* [CS education]	99%
10 (23% of the papers)	1	Bloom, Mesia and Krathwohl (1956) *Taxonomy of Educational Objectives* [education]	100%
Total 239	**176**	**Different Books** (4.1 different books per paper)	

Table 11: The frequency distribution of all books cited in the ICER 2005, 2006 & 2007 proceedings.

4 Books

In this citation analysis, we include as "books" both citations to complete books and citations to chapters within edited volumes, as in our earlier analysis of SIGCSE 2007.

As with ICER citations to conferences and journals the majority of books (82%) were cited by only one ICER paper.

Using the same categorization of books we used for our earlier analysis of SIGSCE 2007, we placed ICER citations to books into one of six categories:

- **Education:** Books that discuss teaching and learning issues in a non-disciplinary specific fashion.
- **Psychology:** Usually educational psychology.
- **Computing Content:** Many of these books were class textbooks, while others were manuals.
- **CS Education:** Books specific to education issues within the computing discipline.
- **Research Methods:** for example, books on statistics, or qualitative research. ICER authors mostly cited qualitative research method books.
- **Social:** Books not concerned specifically with issues in education, psychology, or computing, such as gender issues in the broad context.

Table 10 summarizes our categorization of all books into one of the six categories. The majority of books cited by ICER authors are either concerned with education (28%) or psychology (20%) compared with SIGCSE where the majority (57%) are concerned with computing content. Only 2% of books concerned specifically with computer science education were cited in SIGCSE compared with 17% in ICER.

Table 11 lists the most highly cited books in the ICER proceedings.

5 Age of Citations

Figure 3 shows the number of citations in the ICER papers, for conferences, journals and books, for each year since 1983. Figure 4 shows the same data, cumulatively. Citations drop precipitously for conferences held before 2003, and there are very few citations to conferences held earlier than 1983. Citations to journal papers also drop quickly for papers published before 2003, but not as quickly as conferences. Book citations decline very slowly with age. Inspection of data for the years before 1983 shows a steady trickle of citations to journals and books going back to the 1950s, with a very small number of even older citations. These characteristics of the age of citations are substantially the same as what we observed in our earlier analysis of SIGCSE 2007 papers.

6 Conclusion

ICER authors cite a greater variety of conferences than SIGSCE 2007 authors, who are very focused on the SIGSCE conference series. ICER authors cite more journal papers, from a greater variety of journals, than SIGCSE 2007 authors. In fact, SIGCSE 2007 authors cite comparatively few journals articles.

The most important difference in citations between ICER papers and SIGCSE 2007 papers is in the type of sources that the authors cite. SIGCSE 2007 authors place most emphasis on computing content—curriculum—whereas ICER authors place greater emphasis on citing educational and psychological sources. In our earlier analysis of SIGCSE 2007, we concluded that the SIGCSE 2007 citations suggested an educational epistemology within that community of practice that is primarily objectivist, with the focus on course content. In contrast, our analysis of ICER citations suggests that the education research community is more focussed on students and learning.

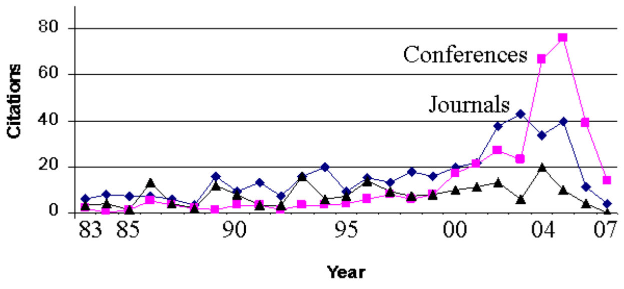


Figure 3: The frequency of citations to books, journals and conferences in the period 1983-2007.

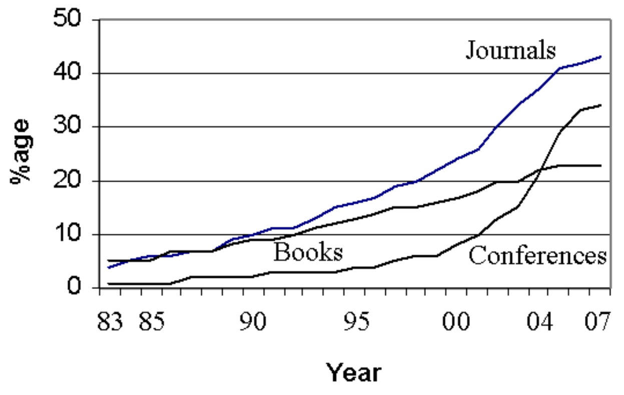


Figure 4: The cumulative frequency of ICER citations to books, journals and conferences in the period 1983-2007.

7 References

ARC (2008) *Excellence in research for Australia initiative, consultation paper*. Available from: http://www.arc.gov.au/pdf/ERA_ConsultationPaper.pdf [Accessed September 2008].

Becher, T. and Trowler, P. (2001) Academic Tribes and Territories. Open University Press; 2nd edition. ISBN: 033520628X

CORE (2007) Computing Research and Education. Available from: http://www.core.edu.au/ [Accessed May 2008].

Lister, R. and Box, I. (2008a). *A citation analysis of the ACE2005—2007 proceedings, with reference to the June 2007 CORE conference and journal rankings.* In Proc. Tenth Australasian Computing Education Conference (ACE 2008), Wollongong, NSW, Australia. CRPIT, 78. Simon and Hamilton, M., Eds., ACS. 93-102. Available from: http://crpit.com/Vol78.html

Lister, R. and Box, I. (2008b). *A citation analysis of the SIGCSE 2007 proceedings.* In Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (Portland, OR, USA, March 12 - 15, 2008). SIGCSE '08. ACM, New York, NY, 476-480.

Lister, R. and Box, I. (2009a) *A citation analysis of the ACSC 2006–2008 proceedings, with reference to the CORE conference and journal rankings.* In Proceedings of the 32nd Australasian Computer Science Conference (ACSW 2009), Wellington, New Zealand, January 2009. CRPIT, 91. Bernard Mans (Ed.), ACS.

Lister, R. and Box, I. (2009b) *A citation analysis of the ADC 2006–2008 proceedings, with reference to the CORE conference and journal rankings.* In Proceedings of the 20th Australasian Database Conference (ADC 2009), Wellington, New Zealand, January 2009. CRPIT, 92. Lin, X. and Bouguettaya, A., Eds., ACS.

Lister, R. and Box, I. (2009c). *A citation analysis of the AUIC 2006–2008 proceedings, with reference to the CORE conference and journal rankings*. In Proc. 10th Australasian User Interface Conference (AUIC 2009), Wellington, New Zealand, January 2009. CRPIT, 93. Calder, P. and Weber, G., Eds., ACS.

Redner S. (1998) How popular is your paper? An empirical study of the citation distribution, *The European Physical Journal B–Condensed Matter and Complex Systems*, 4, 131-134. Available from: physics.bu.edu/~redner/pubs/ps/citation.ps [Accessed November, 2008]

Tsallis C., de Albuquerque M. P. (2000) Are citations of scientific papers a case of nonextensivity?, *The European Physical Journal B–Condensed Matter and Complex Systems*, 13, 777-780. Available from: http://arxiv.org/PS_cache/cond-mat/pdf/9903/9903433v1.pdf, [Accessed November, 2008]

Thomson Scientific (2007) The Science Citation Index Available from: http://scientific.thomson.com/products/sci/ [Accessed May 2008]

Ulrich (2008) Ulrich's Periodicals Directory Available from: http://www.ulrichsweb.com/ulrichsweb/ [Accessed November 2008]

How Students Develop Concurrent Programs

Jan Lönnberg[1] **Anders Berglund**[2,1*] **Lauri Malmi**[1]

[1] Department of Computer Science and Engineering
Helsinki University of Technology,
Espoo, Finland,
Email: {jlonnber,lma}@cs.hut.fi

[2] Department of Information Technology
Uppsala Computing Education Research Group, UpCERG
Uppsala University,
Uppsala, Sweden
Email: anders.berglund@it.uu.se

* Temporary affiliation

Abstract

This paper describes a qualitative, explorative study of how students approach developing and testing concurrent programs. The study is based on interviews with students working on the final programming assignment in a concurrent programming course. We discuss the effects of the students' approaches to constructing and testing programs on their work, how teaching can be improved to support the students in performing these tasks more effectively and how software tools can be designed to support the development, testing and debugging of concurrent programs.

1 Long-term Research Aims

The ultimate goal of our project is to help programmers produce better concurrent programs. Our approach to this is to develop methods and tools, primarily program visualisations, to help programmers understand what a concurrent program does.

Different errors can be the result of completely different ways of thinking. Approaching a problem from the wrong perspective may lead to erroneous conclusions. The nature of the errors can also depend on the perspective or task at hand. Thus, understanding how the programmer is thinking is important in finding ways to prevent errors from being made as well as determining the errors to look for and how to look for them. For example, a programmer who misunderstands the requirements or specification of a system or module or envisions a different purposes for a system will also be testing according to this erroneous understanding of the requirements.

We focus on inexperienced programmers, in particular students, for several reasons. One is that their inexperience means they have more difficulties and therefore need more help. Another is that helping students understand their mistakes not only helps them get their programs to work; it also helps them learn. Thirdly, we can collect and analyse large amounts of data from students, with less effort than from commercial software developers. Finally, it is easier to introduce new ways of working to students than experienced professionals with ingrained habits. We have chosen to focus on studying the understandings of students for three different reasons. Based on the above, our large-scale approach is to first identify the needs of the intended users, students, and then design solutions to address them. The general questions we therefore seek answers to are:

- What kind of defects do programmers inexperienced in concurrent programming introduce in concurrent programs, and why?
- Which of these defects are difficult to locate or understand and why?
- What kind of tools can assist a programmer in finding and understanding these most problematic defects, and how well do they work?

The first results from this project were quantitative information on students' concurrent programs' defects (Lönnberg, 2007). We then proceeded to seek an explanation for the defects we found in the students' understanding of concurrent programming (Lönnberg and Berglund, 2008). This paper describes the continuation of that investigation. Here, we describe students' understandings of the tuple space concept as well as their general understanding of program development and debugging. This study can be seen as exploring the user requirements for future software development tools for students of concurrent programming (the first two questions above). At the end of this paper, we briefly discuss what sort of tools would address the issues by this study (the third question); a theme that we address further in another paper (Lönnberg et al., 2008).

1.1 Aims of This Study

The purpose of the work presented in this paper is to shed light on *how students approach developing and testing concurrent programs.* These insights can serve as a platform for exploring possible sources of errors, especially those that stem from approaches that are ill-suited to developing reliable concurrent programs. The key motivation here is that a better understanding of errors will help us design better software to support program development, understanding and debugging.

In this study, we explore the different ways in which students in a concurrent programming course approach developing and testing a concurrent program. We do this using an empirical, qualitative research approach called *phenomenography* (Marton and Booth, 1997). Phenomenography investigates the qualitatively different ways in which a group of people

Copyright ©2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 95, Margaret Hamilton and Tony Clear, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

experience or think about something. In our study, this provides us with a starting point for exploring many different situations where improvements to teaching or software development tools can be made.

We focus on the design, implementation and testing parts of the software development process. These are the phases in which errors are introduced that can be found by testing performed by developers based on the specification or requirements they are working from, and which require debugging to trace.

Errors in specifying or communicating requirements result in a clearly different form of defect (the program is working as specified), and they can therefore be treated as a separate problem that is not considered here. Similarly, activities performed on the finished code either have no effect on the code as such (e.g. distribution and installation) or can be considered a return to a previous phase (e.g. changing the code to meet new requirements or fixing bugs).

1.2 Students' Understandings of Concurrent Programming

Students may approach the task of developing a concurrent program with different goals in mind than their teachers. Ben-David Kolikant (2005) describes how students understand correctness in a concurrent programming context and how this affects the development process. She notes that students define a "correct program" as a program that exhibits "reasonable I/O for many legal inputs" and that roughly a third of the students were sometimes satisfied with only compiling their program to ensure it is correct.

Considering what students are trying to achieve, it is hardly surprising that they use unsuitable approaches when they develop concurrent programs. Ben-Ari and Ben-David Kolikant (1999) describe how high-school students' concurrent programming conceptions and working methods change during a course on the subject. They found that students have difficulties limiting themselves to operations permitted by the concurrency model, make assumptions based on informal concepts rather than use formal rules and avoid using concurrency and apply development approaches that do not work well in concurrent programming, such as testing a program with a few representative inputs.

One reason for these problems is that students act as users of programs rather than developers. Ben-David Kolikant (2004) describes learning concurrent programming in terms of entering a community of computer science practitioners. She finds that the students initially approach the concurrent programming assignment from a user's perspective, in which only the program behaviour seen through the user interface is taken into account, and not all of them are able to switch to a programmer's perspective.

2 The Study

Two qualitative empirical methodologies, *phenomenography* and an informal qualitative method inspired by grounded theory, are used in this project to explore how students understand concepts in program development. The data for the study are collected through interviews (Subsection 2.2) and are then analysed in two conceptually different ways.

Our key research approach in this project is phenomenography. This approach aims to reveal the different ways in which something is understood by a cohort (Marton and Booth, 1997). In recent years, the interest in phenomenographic research has increased in the Computer Science Education (CSE) community, since the results that are offered, focusing both on the learners and what they learn about, have been shown to be useful within CSE (Berglund, 2006; Berglund et al., 2008). Our use is mostly consistent with this, as we aim to reveal how certain phenomena are understood by a cohort.

Berglund (2006) describes the process of phenomenographic research in computer science education as consisting of a data collection phase and an analysis phase. In the former, the researcher interviews students about the phenomena under investigation.

For most of the questions we investigate, we have found that the students' expressions can be summarised as different perspectives on certain phenomena. In one case, the standpoints expressed by the students do not refer to a single phenomenon but several related phenomena and are therefore better described as a set of differing opinions, less coherent than those that have lent themselves to phenomenographic analysis. Here, we have chosen a different method to list the opinions.

Questions of validity and our responses to them are discussed throughout this paper as they are raised by our methods and results.

2.1 Setting

The students in this study participated in the Concurrent Programming course[1] at Helsinki University of Technology during the autumn of 2006. Students could choose to do the assignments alone or in pairs.

The students were initially required to submit only their Java source code. In the event that their solution was rejected, they were required to submit corrected program code and a report explaining the reasoning behind the erroneous code and the steps they took to correct it.

Lincoln and Guba (1985) emphasise the importance of a natural research setting in getting results that deal with real situations. In this case, the setting was an existing programming course; the only change made for this research was adding interviews. They also argue that determining the transferability of results from one context to another requires knowledge of both contexts; we provide a description of our context to allow the reader to determine which of our results apply to his or her context.

2.2 Interviews

The interviews focused on the third assignment, *Tuple space*, in which the students implement a tuple space (Gelernter, 1985), which consists of a space containing tuples that can be added, read and removed atomically, using Java synchronisation primitives (shared memory, monitors and conditional variables) and use this to construct the message-passing section of a distributed chat server. The students' message-passing code communicates with the rest of the system using method calls; a simple GUI front-end is provided for testing.

The eight students were those who volunteered for the interview out of 16 selected students. Making the interviews a mandatory part of the course for selected students was deemed both an unacceptable demand on the students and counterproductive as the interviews rely on interviewees volunteering information.

While only students who failed the assignment participated in the interviews, this can be seen as purposive sampling (emphasising problems learning concurrent programming). Successful students could

[1]The contents of the course, including the assignments, are described on the course web page: `http://www.cs.hut.fi/Studies/T-106.5600/2006/english.shtml`

have more advanced understandings. However, the understandings described in this paper already cover a wide range from novice to expert understandings, which suggests that also interviewing more successful students may not have affected the results.

In order to maximise the variation of experiences based on the information available to us about the students, we chose groups with different types of problems with their code, as determined by the teaching assistant who graded the assignments. Ten of 31 groups that failed the assignment and two of 24 that passed the assignment were chosen and invited to an interview. Of these groups, seven of the failing groups (six single students and one pair) agreed to participate. The first author conducted interviews with these eight students, after the results for the third assignment were announced and before the resubmission of failed attempts. The focus of the interviews was on the development process, especially the students' reasoning behind their design. The interviews were semi-structured, i.e. they were in the form of a conversation using a set of prepared questions as conversation starters, and lasted from 30 minutes to almost an hour. This allowed students to, in addition to the topics raised by the interviewer, talk about related issues such as their experiences with the other assignments in the course or with programming in a professional context.

2.3 Analysis

The goal of the analysis was to organise the interviewees' utterings into a form that allows the reader to understand the students' different understandings and approaches to developing concurrent programs.

The analysis was done iteratively by the first author in discussion with the second author and, later, also the third author. Each category was intended to represent one understanding or aspect of a phenomenon; the categories were grouped into outcome spaces by the phenomenon they describe. The resulting categories from the last iteration are presented in the following section.

During the analysis it became clear that one of the resulting outcome spaces is not phenomenographic in the sense that it can more obviously be seen as first-order knowledge of what the students have done than second-order knowledge; what the students experience about what they have done. We therefore consider this outcome space to be a grounded theory.

3 Results

In this section we present the results of our analysis. Quotes are used to illustrate the categories. In these, the interviewer is denoted *Int* and the interviewees are assigned, to preserve their anonymity, the names *Evgeniy* and *Elena* (interviewed separately in English), *Filip*, *Fabian*, *Fritjof* and *Frans* (interviewed separately in Finnish) and *Freja* and *Fredrik* (interviewed together in Finnish). The quotes from the interviews in Finnish have been translated into English by the interviewer. Freja and Elena are, as the assigned names indicate, female; the rest are male.

Using quotes from the interviewees allows the reader to see exactly what was said, albeit mostly in translated form. During the interview, follow-up questions were used to clarify the meaning of the interviewees' words where necessary and ensure the interviewer's understanding. The results presented in this paper are a consensus between all three authors. These measures reduce the potential for misinterpretation through e.g. researcher bias.

The phenomena are:

1. **Purposes of the programming task**: the different ways the purpose of the program to be produced in the assignment is understood by students
2. **Sources of failure**: the different entities that may cause failures in the program that the student takes into account
3. **Software development processes**: the overall development process of the students
4. **Approaches to testing**: how the students understand testing their program

The first two describe what the student is trying to achieve by developing a program; students aiming for a passing grade care about correctness in terms of how it affects their grades. Depending on how the student understands the teachers' priorities, this may manifest in different ways.

The latter two outcome spaces are about how the student develops and tests his or her program, like the *Developing and debugging* outcome space from our previous paper (Lönnberg and Berglund, 2008). The categories of *Tuple spaces* in that paper are also connected to different steps in a development process.

Subsection 3.2 is based on grounded theory, while the others are phenomenographic outcome spaces.

3.1 Purposes of the Programming Task

In this subsection, we present what the students perceive as the purpose of their programming task. These purposes are summarised in Table 1.

These understandings are not mutually exclusive; even apparently contradictory understandings can be applied in different contexts by the same person. For example, Fritjof mentions *Assignment (1A)* and *Ideal problem (1B)* as describing how he approached the assignment, but *Possibilities (1D)* as how systems of this type should be written (see below).

The students' aims in a project course in computer systems have been explored by Berglund and Eckerdal (2005). These findings show similarities with the purpose of the programming task discussed in this paper, as they encompass both requirements set by the university and an environment extending the formal requirements, looking toward a professional life.

The different ways in which the student understands the purposes of the programming task are also similar to the *relative correctness* of Ben-David Kolikant (2005) in that students have different understandings of what the program is supposed to be; with the important distinction that relative correctness may involve accepting failure ("The program is correct but it is not finished."), while the purposes of the programming task are alternative interpretations of the goal in which context the program works as intended.

The categories described here can be applied to any programming exercises in an educational setting for which a grade is given. *Assignment (1A)*, in particular, is limited to this context.

1A Assignment In this category, the programming task is understood as a task required to get a grade; i.e. as one task of many required to get a degree. The development process is focused on the demands made by the university setting (such as grading and deadlines) and the correctness or functioning of the resulting program is a secondary concern. The value of the program is understood in terms of how it affects the grade.

One symptom of this is students making design decisions based on how they affect their grade. For

	Label	The purpose of the programming task	What is in focus?	Framework
1A	Assignment	To meet the requirements of the university setting	The university setting's requirements	University setting
1B	Ideal problem	To produce a program that functions within the university setting's requirements	The program itself	University setting
1C	Working solution	To produce a solution to a problem beyond the university setting	The program itself	An environment beyond the university setting
1D	Possibilities	To solve a problem with potential for future development	Possibilities for future development	An environment beyond the university setting

Table 1: Purposes of the programming task

example, when asked why he chose to design his chat system in such a way that it may lose messages under heavy load, Fritjof explains that he "*didn't put much effort into that since it wasn't a factor in failing the assignment*". Ironically, this choice was exactly the reason he didn't pass the assignment.

Decisions about process can also be based on this understanding. When Fredrik and Freja are asked how they intend to resolve a problem found in their program, they explain that lack of time prevents them from solving the assignment properly, so they need to focus on how to pass the assignment.

Similarly, errors are understood in terms of feedback from assistants, as exemplified by Freja expressing how functional her solution seemed to her in terms of expected negative feedback from the grader. Her chat system implementation does not remove closed listening connections at all from the system and thus continues to collect messages that will never be read. Her comment on this is:

> **Freja:** I, for one, thought there were going to be complaints about those `ChatListener`s not being removed.

1B Ideal Problem In this category, the assignment is understood as constructing a program that works in an idealised school environment where practical limitations such as finite memory space and delays do not apply. The focus here is the program itself, albeit as a part of the university setting.

Fritjof provides two different examples of this. He describes how he resolved a problem with messages being lost when listeners do not read them quickly enough by allowing the buffer to expand without limit. He then comments:

> **Fritjof:** Yes, the new solution uses unlimited memory. It's sort of an ideal situation, but isn't this whole assignment a bit ideal?

This shows how Fritjof has changed from one simplification (that a fixed-size buffer will never overflow) to another (that memory is unlimited, so the buffer does not need to have a size limit), and justifying his latter simplification by commenting that the assignment is not representative of reality, so his solution need not be either.

1C Working Solution In this category, the assignment is understood as constructing a program that works correctly in a realistic scenario. The focus is on creating a program that works under normal real conditions (as opposed to the ideal or academic conditions of the previous category), such as the student's own computer. For example:

> **Frans:** I just used some normal use cases and then thought about what problems one could have now and looked if they show up, and if so, why.

1D Possibilities In this category, the assignment is understood not only as a task to be completed, but as a starting point for future development. This category extends the previous one by going beyond the program itself into the realm of possibilities for future software and development.

The possibilities may be better ways to achieve the perceived goals of the assignment. Fritjof explains that the specification prevented him from writing a program that is resilient to network errors, as it did not provide a way to detect network failure. This illustrates how he thinks a chat system should work.

These possibilities may also be ideas for making future projects easier suggested by the assignment. For example, Evgeniy would like to see "*a debugger or some kind of unit testing system for testing synchronised methods*" and justifies this by noting that "*with current tools you end up pretty much proving it on paper*".

Communicating the Purpose of the Assignment to Students

Different aims can easily lead to different types of errors. For example, a solution to an *Ideal problem (1B)* can, as our example shows, lead to a program that fails in practice or is unacceptable from a teacher's perspective. One can argue that students should be warned that the assignment is graded as a *Working solution (1C)* or taught to approach programming in a corresponding fashion. If the assignment is supposed to let the student practise or evaluate the students' skills at determining requirements, this is more satisfactory than explicitly listing everything a student should take into account (e.g. memory use, performance, network failure). This also illustrates how this outcome space could easily have been different if, for example, the grading criteria had been explained in detail and in advance to the students.

When designing systems to explain errors resulting from an unsuitable understanding of purpose, the underlying context must be made clear: the student must understand, for example, that his or her solution fails because there is not unlimited memory, and that this is something that must be addressed in the solution. Visualising memory usage (e.g. by showing object lifetimes and heap allocation over time) may help students understand this type of problem. Another example is assuming the underlying tuple space is always FIFO; giving the student a test setup where this is not true helps expose this assumption.

	Source	Effect on program design
2A	Systems	Tolerate other systems' failures
2B	Programmer	Minimise chances and/or consequences of programmer error
2C	User	Tolerate user error

Table 2: Sources of failure

3.2 Sources of Failure

In order for a program to be useful, it must interact with other entities. This typically involves interacting with other software, especially libraries and operating systems, and human users. As real-world entities tend to be imperfect, risks stemming from the entities can be found that may cause the program to fail.

In the following, we describe the failure sources taken into consideration by students. They are summarised in Table 2.

The three categories here are essentially the three different classes of potentially imperfect entities the student's program interacts with over its lifespan: the student, who as a *Programmer (2B)* may make errors and thus introduce defects, *Systems (2A)*, hardware and software, that the system builds on or interacts with and the *User (2C)*, who may make mistakes when using the program.

A programmer can take any combination of these sources into account. Fritjof shows awareness of the *Programmer (2B)* and *Systems (2A)*, for example, while Filip mentions the *Programmer (2B)* and *User (2C)* (see below).

It should be noted that both the *User (2C)* and *Systems (2A)* sources were de-emphasised by the teachers to avoid complicating the assignment with error checking.

These *Sources of failure (2)* apply to more or less any program (in extreme cases, such as a program with no user input, one of the risk factors may be trivial enough to be ignored).

2A Systems If the goal of the development process is to make a 'bullet-proof' solution, the program must be written to recover from failures in the systems it interacts with. Fritjof's example of *Possibilities (1D)* illustrates this category as well.

2B Programmer An obvious source of failures is defects in the program introduced by the developer.

One reaction to this is to keep things simple.

> **Fritjof:** If I start playing with optimisation, things can go wrong really quickly. I'd break the code that works. Let's just stick to basics.

He goes on to explain how simple structures (in his case, `while(!found)` {try again}) make it easy to show that code is correct:

> **Fritjof:** It was a sort of emergency solution, so you can't get out of the loop before it's really found.

2C User The third and final source of problems is the user, who may provide invalid input or make mistakes. The programmer may, however, have different ideas of what a user error is than the specification. Filip justifies deleting duplicate and empty messages (to compensate for the system occasionally duplicating messages) by arguing that he "*assumed that you don't want to put empty messages*".

Encouraging Awareness of Sources of Failure

The *Sources of failure (2)* considered by the students are closely related in that they also describe the intended context of the program. While the assignment was designed to allow the student to ignore *Systems (2A)* and the *User (2C)* as sources of failure, taking these into account makes sense in a larger perspective and should therefore arguably be encouraged, not discouraged, by teachers. *User (2C)* errors are easy to create (even unintentionally), but simulating problems in *Systems (2A)* can be difficult and is a possible area for testing tool development.

3.3 Software Development Processes

In the previous subsection, we described different ways students had to structure parts of their solution. Here, in contrast, we present different ways they structure their development process, or, in some cases, do not structure. The process understandings are summarised in Table 3.

The six categories of development process models can be seen as a progression from an unstructured or informal development process to a structured one.

Software development is traditionally divided into separate activities that together form a development *process*. The most important of these activities are *requirements specification*, *design*, *implementation* and *verification*. We use this traditional division to describe the students' development process understandings.

In a study of novice programmers, Booth (1992) showed that students approach the task of programming in different ways, varying from cut-and-paste solutions (labelled "Expedient" by Booth) to developing solutions in a structured way, focusing on the problem domain (labelled "Structural"). The results concerning the development process in this project are similar. The differences can be sought in factors such as the differing subject areas, the experience of the students and the development of computer science in the past 16 years.

The process models shown here can, in principle, be applied to more or less any programming project.

3A No Design Needed In some cases, the implementation may be obvious enough from the requirements that the programmer feels no design is needed; no discernible parts can be found in constructing the code from the requirements.

> **Fabian:** The tuple space implementation was done quite mechanically. It was already mostly defined how tuples are put in there and in what form.

Lack of time is another reason to cut a process down to the bare essentials.

> **Fabian:** I didn't think about it, I figured I'd spend very little time on getting it done, what with the deadlines and all.

In this category, only the essential parts of the assignment, the requirements and the end result, the program, are taken into account, and the focus is on producing the end result.

3B Trial and Error Some students showed no signs of having a planned process. Instead, they described their development process as *trial and error*. As in the previous category, there are no discernible intermediate steps in the process. However, a simple structure can be seen: code is repeatedly written

	Label	What is the process understood as?	What is in focus?	Framework
3A	No design needed	Writing code directly based on requirements	Writing code	Requirements and code
3B	Trial and error	Writing code to find a solution that meets requirements	What code works?	Requirements and code
3C	Coding to understand	Writing code to understand the requirements	Understanding the requirements	Requirements and code
3D	Inertia from previous work	Writing code based on own previous work	Writing code	Requirements, code, own experiences
3E	Apply known technique	Using a known technique to structure the solution before implementing it in code	Structuring the solution	Requirements, code, ways to structure code
3F	Adapt known solution	Writing code based on others' previous work	Structuring the solution	Requirements, code, solution archetypes

Table 3: Software development process models

(trial) and found inadequate (error), until the solution is "good enough"

One reason for this is unfamiliarity.

> **Evgeniy:** After a few tries, you realise some things about your previous design and you try to improve it. I wasn't really trying to design it too much ahead, because tuple space was a new thing for me.

Another reason is experiencing problems getting your implementation to work. Frans's approach was "*mostly through trial and error*" after he failed to find a pre-existing solution he could use. He later explains that:

> **Frans:** This trial and error method was a bit bad in the sense that you don't really have a clear picture of what the idea behind it is, how it works, and when you try to fix it, it's kind of hard because you don't know what it's supposed to do, what you have been thinking about.

The distinguishing characteristic of trial and error seen here is lack of planning leading to work that must be redone when it is found inadequate.

3C Coding to Understand While trial and error is often used as a way to reach a solution, misreading or having difficulty understanding the requirements may also lead to trial and error. This category is similar to the last one, except that the purpose of the repeated coding is to understand the requirements. This further separates understanding the requirements (which involves writing code that may solve them) from producing the code that forms the end product.

Some trial and error is inevitable, as one can not understand the development task fully until one has attempted it, making it hard to design ahead.

> **Fredrik:** First I've had to start coding something just to little by little get a sort of grip on what it's about, and not until you start coding the whole thing for the second time does it end up even close.

It's also possible that the student, like Freja, simply "*didn't notice the requirement*".

3D Inertia from Previous Work This category involves solving the assignment the same way as one has done other programming tasks in the past. Compared to the previous categories, a source of approaches, the programmer's experience, is added. This introduces a way to find solutions: reuse your old solutions to old problems.

Fritjof explains that assignments may be "*hard to see as separate entities*" if one has "*previously done the sort of code where you do these big lumps, which is bad*". In his case, he "*just had to put together a monolithic system*".

3E Apply Known Technique Many design techniques such as diagrams exist that can make it easier to design a program. Compared to the previous category, the programmer now has formal design methodology to draw on in solving his or her problem. These are often intended to help split the problem into subproblems by structuring the intended system first.

For example, Fabian "*quickly drew a collaboration diagram containing essentially all the classes worth mentioning*" and then looked at how to build it, in the sense of which message goes where.

3F Adapt Known Solution Instead of unconsciously using one's own previous work as a guide, one can consciously use a known solution as a starting point. Like the previous category, this category adds more knowledge from which a solution can be constructed: other people's solutions to similar problems.

Experiences vary. For example, Frans tried to find something he could adapt from "*some book on concurrent programming [he] borrowed from the library*", but ended up using trial and error, and ended with a problematic solution, of which he said:

> **Frans:** I figured one ought to carefully approach it by thinking of a sort of existing method to use there, since that's the sort of mess where you don't know what it does really.

Fredrik, on the other hand, found:

> **Fredrik:** For many problems, even hard ones, a good and efficient solution has been invented. Usually, even the tough concurrency stuff is abstracted away from the actual business logic that you're implementing.

Why Should Teachers Care about Students' Development Processes?

Many of the process models described here, like the categories of *Development and debugging* in our previous paper, are based on or engender ignorance or lack of understanding, as the students themselves admit. This is worrying considering that the students are not novices at programming. They are also simplistic compared to those described in the software engineering literature, although a complex process is hardly justified in such a small project.

In cases where students structure their solution in well-known ways (*Apply known technique (3E)* or *Adapt known solution (3F)*), it would seem useful to express information about the student's program using the similar structures. If a teacher can infer the structure behind a student's solution, the teacher can explain in the student's terms where (in the process and the solution) the student has gone wrong. Similarly, a student is likely to understand visualisations that show the student his or her program's execution using notation and a partitioning he or she is familiar with.

3.4 Approaches to Testing

In this subsection, we examine the approaches to testing taken by students in terms of what they understand testing to consist of; in other words, how they structure their testing. These views are summarised in Table 4.

Verification, especially in sequential software, typically relies heavily on *testing*. However, the unpredictability of interaction between concurrently executing processes also introduces many pitfalls in the software development process that may result in software defects that are hard to find through testing. There are, therefore, several approaches to ensuring correctness despite nondeterminism, including *deductive proofs* (usually manually constructed) and *model checking*. These formal methods, however, have limited ability to cope with large and complex programs.

Thus, the categories in this subsection can be seen as steps from an undeveloped testing process to a developed one.

Unplanned (4A) testing can be seen as a base that *Breaking the system (4B)* and *Covering different cases (4C)* extend by targeting testing, while *External testing support needed (4D)*, *Testing inadequate (4E)* and *Proof necessary (4F)* are successively clearer pictures of the limits of tests.

While these categories are all applicable to sequential programming, the last two categories are motivated by concurrency issues.

4A Unplanned Unplanned testing involves running the program and passively observing the output to see if anything goes wrong. In this category, input is provided to get the program to do something, but the testing is not directed toward making defects manifest themselves; it's more a matter of convincing oneself that one's program works.

For example, when asked to clarify what he means by "normal use cases" in his quote about a *Working Solution (1C)*, Frans explains that in order to test both tuple space and chat system:

> **Frans:** I just opened two or three chat windows and then wrote some stuff or used the built-in flood feature.

4B Breaking the System The goal of testing can be understood as getting the program to fail, and the testing process then involves setting up cases in which the system is likely to fail. This category extends the previous by adding a goal to the testing: getting the program to fail.

When asked whether he noticed his chat system's failure to enforce message order, Fabian explains the he "*tried to get it to break using the provided user interface*".

Stress testing is one particular way to attempt to break the system. For example, Freja tested her chat system implementation by running many clients and servers with heavy traffic. However, she goes on to explain:

> **Freja:** When I got that running, it worked nicely, so I thought we might even pass this, but, what do you know, there was another "fail".

Her solution relied on the tuple space being FIFO; no testing using Fredrik's (FIFO) tuple space would have exposed this.

4C Covering Different Cases In this category, a strategy for choosing test cases is added: many different cases or ways in which the program can behave are tested. The underlying assumption is that other, untested, cases are similar enough to be covered by these tests. Stress testing is then only part of the cases tested.

Diversity in testing can involve both choosing different data for the program and studying the program's behaviour in different ways. For example, Fritjof says his testing "*was just sort of trying things out with all sorts of cases*". He "*started from the basics*" and moved on to stress testing. Finally:

> **Fritjof:** I went line by line through the lines of code, stopping at certain points in the code and looked at the innards of the program at that point.

Covering different cases does not preclude focusing on likely problems, as Frans explained in *Working solution (1C)*.

4D External Testing Support Needed In this category, limitations of the student's own testing ability appear; the student realises he or she cannot find all his or her own errors and wants outside help.

One reason is being blind to one's own mistakes, like Fritjof:

> **Fritjof:** I found the problem almost directly based on the teaching assistant's explanation. I guess it was a really clear error, and I just couldn't spot it in my own tests; I was blind to that error. That sort of thing is really hard to test without a fancy testing facility or something.

Fritjof also mentions the importance of quick feedback, like in the Goblin (Hiisilä, 2005) programming course management and assessment system he has used in introductory programming:

> **Fritjof:** What I especially like is that you can submit and see what it looks like, red or green, and it sort of gives an impression whether my solution is close to the right one now.

Regarding the API test package provided by the course staff, he says "*That's just 10 % of the assignment; the big problem is the concurrency management*".

4E Testing Inadequate As in the previous category, an awareness of the limitations of testing is added here. In this case, testing in general is seen as insufficient.

	Label	What is testing understood as?	What is in focus?	Framework
4A	Unplanned	Trying out the program to see if it works	How program reacts to input	Features, test inputs and outputs
4B	Breaking the system	Trying to get defects to manifest as failures	Finding inputs that make the system fail	Features, test inputs and outputs
4C	Covering different cases	Trying to show the program can not fail	Finding a set of inputs that gives sufficient reassurance the program will not fail	Features, test inputs, outputs and coverage
4D	External testing support needed	Trying to show the program can not fail, which a programmer cannot reliably do alone	Getting someone else to find a set of inputs that gives sufficient reassurance the program will not fail	Own testing ability and others'
4E	Testing inadequate	Part of ensuring the program is correct	Limitations of testing	Own testing ability and others'
4F	Proof necessary	A complement to a correctness proof	Limitations of testing	Testing and proving correctness

Table 4: Testing approaches

Some defects may be very hard to get to manifest on a normal system. Filip says that in his case all the reasons for failing the assignment were such that they couldn't be found with any decent testing, because "*they were mostly hypothetical*". "*If you're just testing on a home computer, it's really hard to get them to show up*".

In *Covering different cases (4C)*, Fritjof also points out that concurrency-related problems are hard to track with debuggers.

4F Proof Necessary Correctness proofs are considered an important and powerful way to verify a program. This category introduces a solution to the limitations of testing: supplementing it with proofs. Evgeniy's comment about *Possibilities (1D)* is an example of this.

Encouraging Better Approaches to Testing

Many of the students' testing approaches are, like their development processes, superficial and, as the quotes illustrate, easily allow problems, especially related to concurrent programming, to slip through. However, the more advanced testing approaches show an awareness of concurrency-specific factors that affect testing, in particular nondeterminism. As suggested by some of the students, providing testing and debugging facilities that support finding concurrency problems by providing ways to generate and study different types of nondeterministic behaviour would be useful. For example, generating and visualising interleavings both with and without failures for the same input could help students understand why their code is unreliable. The visualisation should emphasise synchronisation, for example by displaying interactions between threads through locking and shared data as e.g. a sequence diagram. Another possible approach is to use static analysis or model checking to search for incorrect synchronisation solutions and point out the relevant parts of the code to the student.

4 Conclusions

In this paper, we present the different ways in which students understand the purpose of concurrent programming tasks, the sources of failure they take into account in doing so, the process models they base their development work on and their approaches to testing. They are found to have a wide range of different understandings ranging from simplistic to advanced. In order to cope with these, teachers should be aware of them and adapt their assignments, grading and tools they provide to the students to take these understandings into account. One of example of such tools would be concurrency testing tools to help students find incorrect assumptions about the execution environment and interleavings that cause failures and visualisations to help them understand their errors and learn by correcting them.

Most of the outcome spaces appear to be applicable to other programming contexts, particularly programming exercises in an educational context. On the one hand, this bodes well for applying the results to other programming courses. On the other, this raises the issue of whether the results say anything about concurrent programming specifically.

4.1 Understanding Goals

The *Purposes of the programming task (1)* and *Sources of failure (2)* uncovered here suggest that many of the errors made by students are misunderstandings of the environment in which their program is expected to run and what it is supposed to do rather than actual misunderstandings of concurrent programming itself. This is in line with our quantitative analysis of students' defects (Lönnberg, 2007). This suggests that teachers should make goals more explicit and provide students with ways to explore problems related to these goals.

Providing students with tools to study memory allocation would help them understand how their programs use (or misuse) memory. This could be as simple as showing them how to use a basic profiler to get information on the maximum memory use of their program. More detailed visualisations, such as charts that show memory use over time categorised by where the memory is allocated, can be used to help students understand memory use in more detail.

Similarly, it is not realistic to expect students to explictly ensure message order if they never see messages get out of order even though they have ignored the issue completely.

The students' *Software development processes (3)* are often disorganised, partially because they do not understand what they're supposed to do. This could also be mitigated by more explicit goals and subgoals or by teaching ways to structure a software development process and a program.

4.2 Generating Test Cases

The students' *Testing approaches (4)* are quite weak. One possible reason is that the students do not take concurrency into account properly in their testing. Another is that, as discussed in the previous subsection, the students do not understand the environment in which their program is to function. This situation could be improved by teaching testing and complementary forms of verification of concurrent programming.

Another approach to helping students test their programs is providing testing tools to generate scenarios that are hard to discover using normal testing procedures. In particular, students need to be able to study how their programs behave when concurrent execution threads interleave in different ways. One possible approach would be to allow students to manually control how their programs' instructions are interleaved, allowing the student to examine known problematic cases in detail. Another is to automatically generate the problematic cases using e.g. a model checker, which helps when students are not aware of a possible problem.

4.3 Understanding Program Behaviour

Debuggers traditionally focus on behaviour on the level of individual statements, as in the implementation category. However, the *Solving technical problems* category (Lönnberg and Berglund, 2008) as well as the *Apply known technique (3E)* and *Adapt known solution (3F)* categories suggest an alternative perspective on debugging: that it would be useful to provide supporting tools, such as execution visualisations, that show program behaviour in ways that support the user's understanding. This could be done by allowing the user to group together parts of the code or execution to correspond to his or her understanding, similarly to the ability to change between program- and algorithm-level behaviour suggested by Price et al. (1993). The tool would then visualise the behaviour of the program in a fashion closer to the programmer's view. For example, if the programmer understands his or her program as a set of communicating entities, the tool should be able to display the communication between these entities and the relevant aspects of their state, even though this state may be spread out over several objects, and part of the communication is implicit in locking mechanisms. Similarly, familiar notation is preferable; if a student has designed a solution using collaboration diagrams, he or she should have less trouble understanding a description of a failure expressed as a collaboration diagram than using an unfamiliar notation.

When communicating a concurrency-related failure to a student, describing the sequence of events leading to the failure can be difficult. Understanding the exact order of events and how this affects the interactions between threads is often crucial to understanding the underlying defect and error and eliminating it. For this reason, the ability to store a particular interleaving for further study is important; this can also be helpful in debugging in general.

4.4 Understanding Errors

The results of this study can also help teachers determine students' errors based on code defects and explanations, by showing the different ways students understand concurrent programming. This is useful when assessing students' work in many ways. One is that it allows grades to reflect the student's understanding and skill better; instead of deducting points based on failures or defects (which may have little to do with the student's skills), they can be deducted for errors that are direct consequences of lack of understanding or skill.

Similarly, the results of this study will help us determine the errors underlying students' defects, allowing more meaningful analysis of these defects. This also helps in explaining the student's defects and errors to him or her.

5 Summary

Many understandings of concurrent programming can be found among students that cause them to write programs that do not work properly:

- Producing a working program is not seen as the purpose of a programming assignment.
- Design is unnecessary or impractical, so developing a program relies on trial and error.
- Testing is cursory and does not take nondeterminism into account.

Our response to these issues is threefold. First, we suggest that students need more explicit and detailed guidance on how to apply different verification techniques in practice and that assignments should be designed to encourage careful development practices. Second, we argue that showing students the consequences of the decisions they make due to their understandings will help them form more useful understandings. Third, some aspects of debugging concurrent programs are difficult, especially testing and debugging. We intend to address this by developing software to help find concurrency-related defects and visualise the failure to facilitate debugging and allow the student to understand his errors and misconceptions.

References

Ben-Ari, M. and Ben-David Kolikant, Y. (1999), Thinking parallel: The process of learning concurrency, *in* 'Fourth SIGCSE Conference on Innovation and Technology in Computer Science Education', Cracow, Poland, pp. 13–16.

Ben-David Kolikant, Y. (2004), 'Learning concurrency as an entry point to the community of computer science practitioners', *Journal of Computers in Mathematics and Science Teaching* **23**(1), 21–46.

Ben-David Kolikant, Y. (2005), Students' alternative standards for correctness, *in* 'The Proceedings of the First International Computing Education Research Workshop', pp. 37–46.

Berglund, A. (2006), 'Phenomenography as a way to research learning in computing', *Bulletin of Applied Computing and Information Technology* **4**(1).

Berglund, A., Box, I., Eckerdal, A., Lister, R. and Pears, A. (2008), Learning educational research methods through collaborative research: the PhICER initiative, *in* Simon and M. Hamilton, eds, 'Proc. Tenth Australasian Computing Education Conference (ACE 2008)', Vol. 78 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society, Wollongong, NSW, Australia, pp. 35–42.

Berglund, A. and Eckerdal, A. (2005), What do our students strive for? Insights from a distributed, project-based course in computer systems, *in* 'Proceedings of 5th Annual Finnish/Baltic Sea Conference on Computer Science Education', pp. 65–72.

Booth, S. (1992), Learning to program: A phenomenographic perspective, Acta Universitatis Gothoburgensis, doctoral dissertation, University of Gothenburg, Sweden.

Gelernter, D. (1985), 'Generative communication in Linda', *ACM Transactions on Programming Languages and Systems* **7**(1), 80–112.

Hiisilä, A. (2005), Kurssinhallintajärjestelmä ohjelmoinnin perusopetuksen avuksi (Course management system for basic courses in programming), Master's thesis, Helsinki University of Technology. In Finnish, abstract in English.

Lincoln, Y. S. and Guba, E. G. (1985), *Naturalistic Inquiry*, Sage Publications.

Lönnberg, J. (2007), Student errors in concurrent programming assignments, *in* A. Berglund and M. Wiggberg, eds, 'Proceedings of the 6th Baltic Sea Conference on Computing Education Research, Koli Calling 2006', Uppsala University, Uppsala, Sweden, pp. 145–146.

Lönnberg, J. and Berglund, A. (2008), Students' understandings of concurrent programming, *in* R. Lister and Simon, eds, 'Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)', Vol. 88 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society, Koli, Finland, pp. 77–86.

Lönnberg, J., Malmi, L. and Berglund, A. (2008), 'Helping students debug concurrent programs', Accepted to Koli Calling 2008.

Marton, F. and Booth, S. (1997), *Learning and Awareness*, Lawrence Erlbaum Associates.

Price, B. A., Baecker, R. M. and Small, I. S. (1993), 'A principled taxonomy of software visualization', *Journal of Visual Languages and Computing* **4**(3), 211–266.

Issues Regarding Threshold Concepts in Computer Science

Janet Rountree **Nathan Rountree**

Department of Computer Science,
University of Otago,
Dunedin, New Zealand
Email: {janet,nathan}@cs.otago.ac.nz

Abstract

Threshold Concepts deserve discussion and reflection in Computer Science Education; they provide a conceptual framework intended to re-empower tertiary educators. At this stage, the idea of Threshold Concepts raises plenty of questions, promises renewed learner and teacher engagement, and suggests a means of focusing on the key aspects of a discipline that will allow a learner to, for example, "think more like a computer scientist." But what precisely are threshold concepts? Can we identify them? Can we agree on which concepts are threshold concepts and which are not? Can we validate them? If threshold concepts do exist, and can be identified and agreed upon, then how would they alter what we teach, how we teach, and how we assess? Do threshold concepts represent anything new or unexpected? The purpose of this paper is to set out issues for the Threshold Concepts model in Computer Science Education and encourage on-going discussion.

Keywords: Threshold Concepts, Computer Science Education, Liminal Space

1 The Notion of Threshold Concepts

The Threshold Concepts model is fashionable. The notion has rapidly gained popularity since being first proposed in 2003 (Meyer and Land, 2003), with the second Threshold Concepts Conference recently being held at Queen's University in Canada (http://thresholdconcepts.appsci.queensu.ca), two books in print (Meyer and Land, 2006a; Land et al., 2008), as well as the publication of many topical articles across a variety of disciplines. The idea has struck a chord with many academics interested in research into the teaching of their discipline and its practice. Examples include Biology (Taylor, 2006), Economics (Davies and Mangan, 2007; Shanahan and Meyer, 2006), Accounting (Lucas and Mladenovic, 2007), Electrical Engineering (Carstensen and Bernhard, 2008), Statistics (Dunne et al., 2003), Geology (Stokes et al., 2007), and Marketing (Lye, 2006). In Computer Science Education (CSE) there is also a developing context for Threshold Concepts (Eckerdal et al., 2006). The purpose of this paper is to discuss work done on Threshold Concepts in CSE, to consider notable issues, and reflect on the usefulness of this conceptual framework to our discipline. For example, we should ask ourselves whether it is possible to agree upon which concepts are threshold concepts and which are not. Can threshold concepts be validated, and how would they alter what we teach, how we teach, and how we assess? We should also consider whether they represent anything new or unexpected.

We can view Threshold Concepts in two parts: first, as a model or framework, and second, as "instance" examples. (To distinguish between instances of threshold concepts, and Threshold Concepts as a model, we shall capitalise the latter.) In the first case, Threshold Concepts provides a model for academics in higher education to develop their teaching and support student learning. The conceptual framework is intended to re-situate teaching and learning within the context of its own discipline, in contrast to the role *learning outcomes* have developed as a managerial tool to audit and monitor "success" (Hussey and Smith, 2003). To contrast the two models, learning outcomes treat education as a set of activities designed to achieve a set of pre-specified outcomes, with success defined in terms of meeting those outcomes. Typically, the outcomes are phrased, "by the end of the course the learner will be able to. . . ." Threshold concepts, on the other hand, state that learners go through a *transformation*, after which they begin to "think more like a computer scientist," and that they gradually acquire the identity of a community of practice. During this transformation, certain parts of the curriculum are pivotal: they represent the "portals" that learners must traverse in order to succeed. To be considered a member of the community of practice, mastery of these concepts is *required*, and the process of mastery is seen as a sort of rite of passage.

Secondly, the term "threshold concept" is used to refer to any part of the curriculum that should be treated as one of these portals. They may be recognised by (probably) being all five of: transformative; irreversible, integrative, bounded, and troublesome (Meyer and Land, 2006b, p7–8). Threshold concepts are transformative in nature because their comprehension creates in the learner a new way of viewing and describing the subject and may alter the learners' perception of themselves and the world. Threshold concepts are irreversible since the fundamental qualitative change that occurs is unlikely to be unlearnt. It follows that threshold concepts are also integrative because learners make new connections, perceive previously unknown relationships, and accordingly change their sense of the world. They are described as bounded, or as boundary makers (Eckerdal et al., 2006, p103), since a threshold concept "...helps to define the boundaries of a subject area because it clarifies the scope of a subject community" (Davies, 2006, p74). The final characteristic is that threshold concepts embody knowledge that is troublesome for learners to grasp—it is more than simply new subject matter, it is material that is diffi-

Copyright ©2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 95, Margaret Hamilton and Tony Clear, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

cult and possibly counter-intuitive, and accordingly it cannot, as yet, integrate with the learner's current mental schema.

The two most important of the five characteristics of threshold concepts are the troublesome and transformative nature of the knowledge. Troublesome knowledge goes beyond knowledge that is difficult to understand—it is tied up with incorrect or incomplete mental models, misconceptions, the inability to transfer understanding from one context to another, conflicts with current understanding or perspective, emotional response, (e.g. being vexing), and tacit presumptions (Perkins, 2006). Working through problems and gaining understanding of troublesome aspects of the subject matter is interpretative: it extends the use of language; modifies ways of thinking and practising; it is a process of "identity formation" through which the learner gains entrance into the subject community (Davies, 2006). The Threshold Concepts framework views "... learning as a form of journey, during which the student not only gains insights great and small, but is also changed as an individual by new knowledge" (Meyer and Land, 2007). In our case, the learner begins to think and practice 'more like a computer scientist.'

All threshold concepts are also core concepts, but not all core concepts are threshold concepts. For example, the ACM "Curriculum Guidelines for Undergraduate Degree Programs in Computer Science" presents a body of knowledge including core topics. Picking topics arbitrarily, "object-oriented programming" may be a threshold concept, but the "history of computing," whilst recognised as a core topic, may not be a threshold concept. The "history of computing" (in this argument) might be new knowledge, it might be complex knowledge, it might require effort to learn, but it is not troublesome or transformative in nature for the learner.

Let us take recursion as a potential example of a threshold concept in CS: does recursion meet the five defining characteristics? We expect that many CS educators would agree that recursion presents an example of troublesome knowledge. When first seen the notion of 'self reference' is alien and many novice programmers struggle to come to terms with this concept in implementation and description. Once a learner 'gets' recursion she has a significant transformation in her mental process. For example, she sees recursive sets rather than the state a program gets into, ways of describing program state change to include base/stopping cases related to the atoms of a recursive set. Elegance in program design is brought into focus. She starts to make connections and see relationships with other material such as the fact that all loops can be expressed as recursion. It is unlikely that this new understanding will be unlearnt, so it follows that recursion is irreversible as well as being integrative. On the one hand recursion is useful in the practice of programming, on the other hand it is a theoretical construct that defines what is computable; hence recursion is a boundary marker for both Software Engineering and Theoretical Computer Science.

2 Liminal Space and Pre-liminal Variation

One of the most powerful inferences that can be drawn from Threshold Concepts is that of *liminal space.* Meyer and Land suggest that a threshold concept is rarely mastered in a single "aha" moment, but instead requires a period of time over which a student makes the transition. The period of transition is referred to as "liminal space" (from the Latin *limen*, meaning boundary or threshold). Students who are in the period of transition may be characterised as undergoing a "rite of passage," at the end of which they will have achieved new knowledge and status within their community. The rites may be drawn-out, confusing, and require that students begin to think and act differently to be seen as having succeeded. (Success here is defined as not "understanding how a practitioner thinks," but "beginning to think like a practitioner.")

The proponents of Threshold Concepts characterise liminal spaces as the places where students "get stuck" if they are going to get stuck at all. Students show such a range of ability to traverse the liminal space that it is natural to think of them as being "effective" or "ineffective" at negotiating the liminality. Meyer and Land suggest therefore that *pre-liminal variation* is the key to understanding how and why students might effectively negotiate liminal space. What is there in each student's background that might help or hinder their liminal journey? In examining pre-liminal variation, we may need to attend to more than just whether or not they have mastered the academic material considered to be pre-requisite; perhaps also we need to examine their epistemological stance: are they ready to build knowledge in the way we expect them to, and will they be able to tolerate the (possibly quite long) period of uncertainty, confusion, and even oscillation between seeming to have "got it," and feeling sure that "it" will remain forever elusive?

Eckerdal et al. (2007) present research to support the notion that students in Computer Science can be accurately characterised as spending time in liminal space. Interviews with students regarding things that they had previously identified as potential threshold concepts established that all of the proposed features of liminal spaces were evident: significant time commitment, oscillation between states, emotional involvement of anticipation and anxiety, and mimicry of the new state.

The implications attendant upon *liminal space* may well prove as significant as the Threshold Concepts model itself. First, Eckerdal et al. note that, contrary to popular conceptions of "levels" of understanding, students passing through liminal space cope with different aspects of concepts (theoretical, practical, etc.) in parallel. Second, that the *time* required to make the transitions is significant, and perhaps unexpected to novice students (and perhaps to those setting learning outcomes). Third, that there is a significant *emotional* reaction to dealing with liminality, and that such reaction is normal and should be managed rather than ignored or dismissed. Finally, that mimicry during the negotiation of liminal space may not be undesirable, but may be a normal part of the process of coming to terms with conceptual difficulty.

3 Identification of Threshold Concepts

Assuming that the Threshold Concepts model is valid, what processes can we use to identify threshold concept instances? Davies (2006) notes that threshold concepts provide a method of describing the 'way of thinking' distinctive to a discipline; a method that is an alternative to the 'key concepts' idea or the method of phenomenography. However, he also notes that identification of threshold concepts may be difficult due to their being "taken for granted" within a subject, and "therefore rarely made explicit." He goes on to suggest two methods for recognising the threshold concepts within a discipline. (For convenience we shall refer to them as the *first approach* and the *second approach.*) The first approach argues that we might recognise threshold concepts by examining the different ways in which two disciplines analyse the same situation. For instance, if Social Scientists analyse school choice as a zero-sum game, but Economists as a problem of general equilibrium, that might lead us to believe that equilibria represent a threshold concept in Economics.

The second approach to identifying threshold concepts described by Davies is to focus on the distinction between people inside and outside the community of practice—specifically, on the differing ways in which students and experts in the field analyse the same problem, or group of problems. This, of course, is empirically very convenient for educators in a given field, as they have the best opportunities to conduct research on their own students. Consequently, most work on identifying threshold concepts within disciplines has focused on this approach. The advantage is that it allows researchers to look at problems that only exist within one field. The clear disadvantage is that there is no equivalence between novice/expert comparisons and expert/expert comparisons.

Most substantial work on identifying threshold concepts in Computer Science has taken the *second* approach, examining the responses of *students* in Computer Science to questions about where they got "stuck" while studying. (See the comments on studies in Computer Science in Section 5.) To date, the first approach (examining the differences in methodology between related fields) has been largely ignored. This seems a missed opportunity, since Computer Science has a wealth of related fields, many of which have shared interests. Examining the different ways in which practitioners in Computer Science, Information Systems, Mathematics, Physics, Electrical Engineering, and Linguistics, tackle similar problems may produce excellent candidates for threshold concepts in each discipline, and opens up a research question concerning whether threshold concepts are shared between disciplines (and thus whether there is a hierarchy of threshold concepts), and whether threshold concepts mutate as they cross between disciplines.

Work has already begun on validating the Threshold Concepts model and on identifying instances of threshold concepts in Computer Science, e.g. in Eckherdal et al.'s multi-national study. This research has made it clear that students in Computer Science encounter things that look much like threshold concepts and liminal spaces, but that there are difficulties in articulating the *granularity* of such things. For instance, both lecturers and students referred to "object-orientation" as a threshold concept, but the authors note that this is almost certainly too broad a term, when interviews reflected that the "stuck places" were more at the level of polymorphism or object cooperation. Work on curriculum design by Mead et al. (2006) may provide a way of teasing apart these hierarchical distinctions; they propose the idea of an *anchor concept* (which is a concept that is either foundational or transformative AND integrative) and an associated anchor concept graph, which maps the cognitive load shared by related anchor concepts. This approach might allow us to specify concepts at multiple levels of granularity, and in a logical order that recognises the dependency of threshold concepts on other foundational material.

Some logical difficulties in identifying threshold concepts have been identified. Rowbottom (2007) raises several general caveats, all of which apply to threshold concepts in Computer Science (and, indeed, any other subject). His notes of caution are as follows:

1. The features attributed to threshold concepts are insufficiently precise to distinguish them from any other concept. They are described by their originators as *probably* and *not necessarily* transformative, irreversible, troublesome, etc. Thus, any concept you care to mention might be a threshold concept, even though it has none of the features, and any concept that has all of the features may not in fact be a threshold concept. Thus, I may argue that "scope" is a threshold concept in Computer Science, and you may argue that it is not, but neither of us can properly appeal to the definitions to support our argument. Without those definitions, it is not logically feasible even to use empirical research to support or refute a claim that something is or is not a threshold concept.

2. There are at least three accounts of what constitutes a *concept*, from Cognitive Science (mental models functionally equivalent to symbols or words, complete with combinatorial syntax and semantics), to competing views in Philosophy (the concept of X is reducible to the *ability* to think of Xs or classify things as Xs; or concepts as abstract entities of thought associated with names). Possibly these views are not contradictory, and possibly we are meant to default to the view of Cognitive Science. But which view we hold will profoundly affect our method of determining whether a particular concept has been mastered. Rowbottom presents "playing tennis" as an example of an activity where there is a distinction between *knowing that* and *knowing how*; we might just as readily suggest that programming is an activity where changes in concept do not necessarily result in changes of practice, nor that changes in practice are guaranteed to be a result of a change in concept.

3. Not only are the qualifiers attached to each feature a problem, but so too are the features themselves. Take "transformative" as an example. What may be transformative for me, may not be transformative for you. For instance, if I learned Pascal as my first programming language, then the notion of *generics* would tend to have all the features of a threshold concept—in particular, troublesome, transformative, and integrative—because Pascal does not have the features necessary to support truly generic container types (specifically, untyped pointers, or the ability to specify that all the objects in the container will be the same size if not the same type). [1] In contrast, a student who has Java as a first language is likely to have no difficulty at all with the idea that a container can store things of many types, since containers can always be defined as storing things of type "Object." Thus, *genericity* is unlikely to have any of the connotations of a threshold concept to a student who has Java as her first programming language.

These caveats are not necessarily enough to dismiss the Threshold Concepts model, nor the possibility of identifying good candidates for threshold concept instances. Even though our definitions of threshold concepts may not be perfectly precise, we can defeasibly posit their existence, and agree upon their most distinctive features, until such time as we find evidence to suggest that we should retract our assertion. Imprecise definitions are insufficient evidence for retraction; "four-legged mammals" might be an imprecise definition for "cats", but that does not imply that cats do not exist. However, these problems

[1] Variant records will allow a programmer to contain a set of specified types, but will not allow the containment of any new type without modification of the record. Furthermore, if one record holds things of type A or type B, code that processes the record has to deal with the possibility that type A processing **can** be invoked on type B objects.

with the identification of threshold concepts do indicate that there is only a limited amount of utility in trying to determine threshold concepts by empirical means. Science, along with its attendant methods and expectations, is a social construction: if there are threshold concepts, and they are really as critical as the model claims, it is because *we* have put them in place and made them so. Studying our students for signs of threshold concepts may help to make our tacit assumptions explicit; but studying differences in practice between experts in related fields should provide a broader set of concepts that act as boundary markers for a subject.

4 Consequences of Threshold Concepts

Land et al. (2006) state that the Threshold Concepts idea "presents important challenges for curriculum design and for learning and teaching." Specifically, they draw attention to nine considerations that they feel are important with respect to the design and evaluation of curricula. These are:

1. that threshold concepts are the "*jewels in the curricula*," demanding longer and sharper focus than other concepts;

2. that they emphasise the *"importance of engagement"* by stressing the transformation of the student into someone who thinks like a computer scientist, rather than someone who understands how computer scientists think;

3. that they emphasise *listening for understanding* on the part of the teacher—in particular, listening for the signs of pre-liminal variation among students;

4. that they emphasise the *reconstitution of self* on the part of the student, and imply the design of an environment supportive to the discomfort of repositioning oneself in relation to the subject;

5. that they provide good reasons for *tolerating uncertainty* both on the part of students and teachers, due to the time it takes to negotiate the liminal space of a Threshold Concept;

6. that they promote *recursiveness and excursiveness* of learning—that troublesome knowledge often requires re-visiting, and that the "outcome" of learning is not just a set of "the learner will be able to. . . " statements, but that the learner will have been transformed by the journey into one who thinks differently;

7. that understanding *pre-liminal variation* among students will help us to understand why some students negotiate the curriculum effectively, while others have more difficulty;

8. that they may expose the *unintended consequences of generic 'good pedagogy,'* in that they provide examples where standard methods (such as simplifying the concept to begin with) prove dysfunctional;

9. that they highlight an aspect of the *underlying game*—that is, that students will only become members of the community of practice if they master the *authorised* understanding of threshold concepts, and that alternative versions (based on personal experience or common-sense) will place them in unwitting opposition to the community.

At first glance, it is easy to dismiss these considerations as "nothing new here." After all, in Computer Science, we are only too well aware that some topics need longer and greater emphasis than others, that student engagement is paramount to success, that student variation is immense and requires significant adaptability on the part of teachers. We know all of these things, and using the language of Threshold Concepts to express them is unlikely to affect our understanding of them, nor provoke a radical shift in our strategies for dealing with them.

However, there remains the possibility that the Threshold Concepts model may provide *unexpected* consequences. For instance, we can reason as follows: threshold concepts are integrative; they allow the practitioner to combine other fundamental concepts in ways unique to the discipline. If we believe object-orientation to be a threshold concept, what other fundamental concepts are being integrated? A strong implication of the Threshold Concepts model is that those concepts that are to be integrated should be grasped first by learners, before attempting to traverse the liminal space where they will learn to integrate them. If, for instance, we view the encapsulation of state and behaviour as a key part of object-orientation, the implication is that state and behaviour should be mastered first. This suggests that an objects-first approach to CS1 is incompatible with the Threshold Concepts model. To date, there has been little or no work on exploring the consequences of Threshold Concepts in this fashion: what teaching practices would we adopt, and what should we reject if the Threshold Concepts model is valid?

At a higher level of abstraction, is it possible—or even necessary—that we can ever come to a general agreement on what constitute threshold concepts in Computer Science? It could be argued both ways: that we can and should, and that we cannot and that it does not matter. For the latter argument, we need only point to the ACM curriculum and state that some topics will be troublesome for some learners, and different topics will be troublesome for other learners. We adapt as necessary, listening carefully to our classes for signs of difficulty and for indications of dawning mastery. In adopting that attitude, the Threshold Concepts model tells us nothing new, leads to no insights, and has no implications. On the other hand, what happens to an academic subject when the *underlying game* (a phrase used often in Threshold Concepts literature) remains implicit, and is never made explicit? Even in Economics, where different schools of thought (such as the Austrian School, or Keynsian Economics) provide very different analyses of the same events, they all agree that they are *doing economics*, based on a shared set of underlying, unifying concepts.

Those concepts *can* be identified by focusing on what we do that is peculiar to our discipline, and by making explicit those things that we think everybody agrees on. Some of that identification can no doubt be achieved by observing students making the transition from not thinking like a practitioner to doing so, and some can be achieved by observing how practitioners in different but related fields practice differently. It seems reasonable that the two different methods will highlight different concepts.

5 Studies in Computer Science

Shinners-Kennedy (2008) proposed *state* as a threshold concept in Computer Science, showing that it meets all five threshold concept criteria. Vagianou (2006) considers *program-memory interaction* as a possible example of a threshold concept in Computer

Science. The author argues that research shows a viable computer model must be present before programming is engaged, and that an introductory programming course needs to shift each student's viewpoint from that of a non-expert, "end-user" stance toward a "programmer stance" with an awareness of being directly responsible for the computing process. Discussion suggests that Program/memory interaction displays the characteristics of a threshold concept. The notion is troublesome, since beginning students do not realise *how* the use of memory takes place, or their role in that process. It acts as a boundary marker because our biological concept of short and long term memory differs from computing memory. It is integrative, showing otherwise hidden relationships between hardware and software. Program/memory interaction is transformative, since once understood it will significantly shift the student's perspective and it follows that the notion should be irreversible since the knowledge is very unlikely to be unlearnt.

A paper by Khalife (2006) aims to identify potential Threshold Concepts in introductory programming courses and propose solutions to help students surpass thresholds. The author presents some commonly accepted novice programmer difficulties (such as lack of problem solving strategies), and then suggests that the first threshold a student needs to pass is "...to develop a simple but yet concrete mental model of the computer internals and how it operates during program execution" (Khalife, 2006, p246). A computer model for teaching purposes is then set out along with results of an empirical evaluation of that model.

To date, the most systematic and in-depth approach to studying Threshold Concepts in Computing is an on-going multi-institutional, multi-national series of projects underway in the UK, USA, and Sweden (Zander et al., 2008; Eckerdal et al., 2006; Boustedt et al., 2007; Eckerdal et al., 2007; Moström et al., 2008). Discussions by members of this group have been underway since (at least) early 2005 when, at the Conference on Innovation and Technology in Computer Science Education in Portugal, they "...interviewed 36 Computer Science educators from nine countries and asked for suggestions about concepts that met the criteria for a threshold concept" (Zander et al., 2008). There was no universal consensus of concepts amongst academics, but the most popular are listed as: levels of abstraction; pointers; the distinction between classes, objects, and instances; recursion and induction; procedural abstraction; and polymorphism. At the 5th Koli Calling conference in Finland, McCartney and Sanders (2005) presented a poster on Threshold Concepts in CSE and collected opinions from delegates on potential Threshold Concepts through questionnaires and interviews. At ITiCSE'06, the authors provided a helpful discussion of related areas of research in CSE, namely: constructivism; mental models; misconceptions; breadth-first approach to teaching; and, ideas fundamental to the discipline (Eckerdal et al., 2006). The paper continues by proposing, with support from literature, two candidates for threshold concepts—abstraction and object-orientation. A paper presented at *SIGCSE'07* answers *yes* and *yes* to the title "Threshold Concepts in Computer Science: Do they exist and are they useful?" (Boustedt et al., 2007). The principal contribution of this paper is to describe their empirical approach of using structured interview techniques to identify candidate threshold concepts in Computer Science. From 33 concepts mentioned by students and educators, they examine two in detail to establish that they have the required features: object-orientation and pointers. Here, they make the point that what they might have uncovered are "...perhaps broad areas in which thresholds exist." Interview subjects were inclined to use the broad terms to identify the concepts, but then spoke in much more specific terms about problems they encountered within those areas.

What the on-going multi-national study provides is validation of the Threshold Concepts model for CSE. Possible "instance" examples of threshold concepts (which display the five threshold criteria), have been identified by practitioners from different countries. CS students were interviewed regarding topics they had found troublesome and "got stuck" on, and an intersection of topics identified by both (object-oriented programming and pointers) have been investigated in greater detail for evidence that they satisfy the threshold concept criteria. The multi-national study has also looked at the idea of liminal space as an appropriate description of the transitional space CS students negotiate as they develop ways of thinking and practising.

6 Concluding Comments

The Threshold Concept Model presents a disciplinary situated learning framework for higher education which is a welcome shift in perspective away from the checkbox flavour of learning outcomes. "Instance" examples of threshold concepts are core curriculum concepts with the particular properties of being transformative, irreversible, integrative, boundary markers, and troublesome. In Computer Science, proposed threshold concept examples include: state; program-memory interaction; levels of abstraction; pointers; the distinction between classes, objects, and instances; recursion and induction; procedural abstraction; and polymorphism. With further work, is it likely that academics in CSE would agree on a set of threshold concept topics? Perhaps! We think it likely that there will be agreement for some threshold concepts and not others. In the empirical sense, it is not possible to validate threshold concepts because what is a threshold concept for one person, may not be for another. The best we can achieve is a sense of "many" or "most" learners finding a particular topic meets the requirements of a threshold. Does this lack of validation matter for CSE? No, we think not, because practitioners define the subject; we define the curriculum (not the students), and so empirical validation using students may be something of a "red herring." If threshold concepts define the boundaries of how practitioners perceive a subject, then surely we need to study practitioners if we wish to define the threshold concepts.

If practitioners have differing perspectives then you simply get different schools of thought. Will threshold concepts alter what and how we teach and how we assess Computer Science? Yes and No. On one hand, for example, object-orientation is troublesome and counter-intuitive (we already know this and give it extra teaching emphasis) so labelling it a threshold concept provides no additional enlightenment. On the other hand, investigating students' *liminal space* as they come to terms with object-orientation will provide valuable insight into what makes an effective novice. Knowing what makes a novice effective in traversing liminal space allows particular skills and ways of thinking to be targeted. Discovering an unexpected threshold concept would be of much interest. The implications of threshold concepts are also of interest for CSE; for example, is an objects-first approach in teaching incompatible with Threshold Concept theory? Threshold Concepts literature strongly suggests that the integrative nature of thresholds requires students to first have mastered some fundamental concepts before embarking on the threshold concept itself. Simplification of a threshold concept to make it easier has been shown (in Economics at least) to lead students to settle for a naive version of that knowledge (Land et al., 2006, p333). An implication for CSE could be seen as avoiding teaching "objects-first" in CS1.

From a philosophical viewpoint the difficulty with Threshold Concepts is the lack of a formal definition which would allow specification of what is, and what is not a threshold concept. The model needs to become more precise because, as it stands, no threshold concept candidate can be verified—it is only possible to make an assertion that it exists. We suggest that one small step to help clarify the model is to view Threshold Concepts in two parts: first, as a model or framework, and second, as "instance" examples. Both need to be validated separately—are there subjects for which the Threshold Concepts model is completely invalid? Or does it work for every tertiary subject? If you have a subject for which Threshold Concepts is a valid model, what are the threshold concepts within it? How can you tell for sure? How can you tell if you have identified all of them? Is it possible to distinguish those things that look like threshold concepts but are not?

Although we have titled this paper "*Issues with Threshold Concepts in Computer Science*," and have noted some difficulties, we do not think that the notion of Threshold Concepts should be dismissed. As educators in Computer Science there are essential questions which we continually ask: When do we consider a student has been successful? Why are some students successful, whilst others are not? And how do we support students through their learning process? In relation to these questions, there are three aspects of particular value that Threshold Concepts contribute. Threshold Concepts provide:

1. an apt description for what it means for a student to have been successful in our discipline;
2. the addition of epistemological concepts of *liminal* and *pre-liminal space*, which gives direction for future research into what makes an effective novice programmer;
3. a focus on our *community of practice*, giving deference to the disciplinary knowledge of the academic, so that concerns are "... always analysed and resolved from, and within, specific and situated disciplinary contexts" (Meyer and Land 2007, p14).

Where should the Threshold Concept discussion for CS education go next? We suggest that Davies' *first* approach to identifying threshold concepts—examining the ways in which practitioners in related disciplines solve similar problems—provides the best avenue for further research. If the goal is to identify "how to think like a Computer Scientist" then we must first study the practitioners, not their students. In CS the first challenge is to specify what constitutes our subject. If we ask our colleagues what it means to be a Computer Scientist, how much agreement will there be? For example, how much overlap will there be between groups with a software engineering flavour, or theoretical, or electrical engineering approach? The immediate value of Threshold Concepts in CS Education is to require us to address what it means to be a Computer Scientist.

References

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., and Zander, C. (2007). Threshold concepts in computer science: do they exist and are they useful? *SIGCSE Bull.*, 39(1):504–508.

Carstensen, A.-K. and Bernhard, J. (2008). *Threshold Concepts and Keys to the Portal of Understanding: Some Examples from Electrical Engineering*, pages 143–154. In (Land et al., 2008).

Davies, P. (2006). Threshold concepts: How can we recognise them? In (Meyer and Land, 2006a), pages 70–84.

Davies, P. and Mangan, J. (2007). Threshold concepts and the integration of understanding in economics. *Studies in Higher Education*, 32(6):711–726.

Dunne, T., Low, T., and Ardington, C. (2003). Exploring threshold concepts in basic statistics, using the internet. In *AISE/ISI Satellite*, Berlin.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., and Zander, C. (2006). Putting threshold concepts into context in computer science education. *SIGCSE Bull.*, 38(3):103–107.

Eckerdal, A., McCartney, R., Moström, J. E., Sanders, K., Thomas, L., and Zander, C. (2007). From limen to lumen: Computing students in liminal spaces. In *ICER '07: Proceedings of the Third International Workshop on Computing Education Research*, pages 123–132, New York.

Hussey, T. and Smith, P. (2003). The uses of learning outcomes. *Teaching in Higher Education*, 8(3):357–368.

Khalife, J. T. (2006). Threshold for the introduction of programming: Providing learners with a simple computer model. In Romero, P., Good, J., Acosta, E., and Bryant, S., editors, *Proceedings of the 18th Workshop of the Psychology of Programming Interest Group*, pages 244–254.

Land, R., Cousin, G., Meyer, J. H., and Davies, P. (2006). Implications of threshold concepts for course design and evaluation. In (Meyer and Land, 2006a), pages 195–206.

Land, R., Meyer, J. H., and Smith, J., editors (2008). *Threshold Concepts Within the Disciplines.* Sense Publishers.

Lucas, U. and Mladenovic, R. (2007). The potential of threshold concepts: An emerging framework for educational research and practice. *London Review of Education*, 5(3):237–248.

Lye, A. (2006). Threshold concepts: Reflections on marketing education. In *Proceedings of the 2006 ANZMAC Conference*, Brisbane.

McCartney, R. and Sanders, K. (2005). What are the "threshold concepts" in computer science? In *Proceedings of the 5th Baltic Sea Conference on Computing Education Research (Koli Calling 2005)*, page 185.

Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. S., and Thomas, L. (2006). A cognitive approach to identifying measurable milestones for programming skill acquisition. *SIGCSE Bull.*, 38(4):182–194.

Meyer, J. H. and Land, R. (2003). Threshold concepts and troublesome knowledge – linkages to ways of thinking and practising. In Rust, C., editor, *Improving Student Learning Theory and Practice – Ten Years On*, pages 412–424. OCSLD, Oxford.

Meyer, J. H. and Land, R., editors (2006a). *Overcoming Barriers to Student Understanding: Threshold Concepts and Troublesome Knowledge*. Routledge.

Meyer, J. H. and Land, R. (2006b). Threshold concepts and troublesome knowledge: An introduction. In (Meyer and Land, 2006a), pages 3–18.

Meyer, J. H. and Land, R. (2007). Stop the conveyer belt, I want to get off. *Times Higher Education Supplement*, page 14. Issue 1807, 17 August 2007.

Moström, J. E., Boustedt, J., Eckerdal, A., McCartney, R., kate Sanders, Thomas, L., and Zander, C. (2008). A multi-national, multi-institutional project on threshold concepts in computer science: Results and implications. In *Threshold Concepts Conference 2008*, Queen's University, Kingston, Canada.

Perkins, D. (2006). Constructivism and troublesome knowledge. In (Meyer and Land, 2006a), pages 33–47.

Rowbottom, D. P. (2007). Demystifying threshold concepts. *Journal of Philosophy of Education*, 41(2):263–270.

Shanahan, M. and Meyer, J. H. (2006). The troublesome nature of a threshold concepts in economics. In (Meyer and Land, 2006a), pages 100–114.

Shinners-Kennedy, D. (2008). The everydayness of threshold concepts: 'State' as an example from computer science. In (Land et al., 2008), pages 119–128.

Stokes, A., King, H., and Libarkin, J. (2007). Research in science education: Threshold concepts. *Journal of Geoscience Education*, 55(5):434–438.

Taylor, C. (2006). Threshold concepts in biology: Do they fit the definition? In (Meyer and Land, 2006a), pages 87–99.

Vagianou, E. (2006). Program working storage: A beginner's model. In Berglund, A. and Wiggberg, M., editors, *Proceedings of the 6th Baltic Sea Conference on Computing Education Research (Koli Calling 2006)*, pages 69–76.

Zander, C., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., and Sanders, K. (2008). Threshold concepts in computer science: A multi-national empirical investigation. In (Land et al., 2008), pages 105–118.

A Taxonomic Study of Novice Programming Summative Assessment

Shuhaida Shuhidan **Margaret Hamilton** **Daryl D'Souza**

School of Computer Science & Information Technology
RMIT University,
PO Box 2476V, Melbourne 3001, Australia,
Email: {shuhaida.mohamedshuhidan, margaret.hamilton, daryl.dsouza}@rmit.edu.au

Abstract

Learning to program is difficult, a situation that is largely responsible for high attrition rates in Computer Science schools. Novice programmers struggle to grasp an early understanding of programming, which can lead to frustration and eventually surrender. The problem has generated interest in a range of enquiries, and has given impetus to the need for a teaching-research nexus towards a better understanding of novice programming problems. We continue the trend in this paper and report on a study we have conducted of novice programmers' efforts in summative assessment. Our study involves multiple-choice questions and coding question drawn from a programming examination. We analyse the answers provided by novices to final examination questions, and attempt to understand why students make such errors. We aim to categorise and classify the questions in the context of two well-known learning taxonomies: Bloom's Taxonomy and the SOLO Taxonomy.

Keywords: Summative Assessment, Programming Errors, Novice Programmer, Bloom's Taxonomy, SOLO Taxonomy, Taxonomy of Programming

1 Introduction

High attrition (or low retention) rates are often experienced in Computer Science schools (Bennedsen & Caspersen 2007), in part due to students' inability to learn programming (Oman et al. 1989), a core survival skill. Novice programmers are often known to have difficulty in grasping the foundation level programming concepts sufficiently early, resulting in grief and frustration, and ultimately, surrender. Yet, those who do manage to overcome their learning difficulties are able to move on, even excel.

A *novice programmer* (henceforth referred to simply as a *novice*) is in the first stages of being a programmer (Bonar & Soloway 1983, Thomas et al. 2004) and also has been described as an end-user who wants to program a computer (Smith et al. 2000). Novices range from being those who have never previously experienced programming to those may have some basic background to programming, attained informally or via formal study in pre-university contexts. In our study, we restrict our definition of novices to tertiary level students who are learning programming for the first time.

Copyright ©2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 95, Margaret Hamilton and Tony Clear, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Clear et al. (2008) point out that two major contributors to the exodus from Computer Science and cognate disciplines are poor teaching of computing in high schools and the "nerdy" image of the profession to young people. We contend that, even at university level, too much is taken for granted by teachers, in the formative stages of learning programming. Incorrect models of a range of fundamental programming concepts are conveyed in teaching or received by novices, often compounding the problems of understanding. Poor models of programming concepts have a propagating effect, plunging novices into a spiral of frustration, loss of confidence and self-belief, as more complex material is covered (Wayne D. Gray 1993, Caspersen & Bennedsen 2007).

In this paper we report our own efforts to develop a better understanding of learning difficulties in programming. Our study analyses the novices' final examination answer sheets and attempts to classify the questions and responses in terms of two taxonomies of learning: Bloom's Taxonomy (Bloom 1956) and the SOLO Taxonomy (Biggs & Collis 1982) (SOLO is an acronym for Structure of the Observed Learning Outcome). First, we study the multiple-choice questions and responses in the examination and classify them according to Bloom's Taxonomy.

We extend our study to look at a programming code question to explore the understanding displayed by novices in writing segments of code. We classify these coded responses using the SOLO taxonomy. Next, our classification of responses according to the two taxonomies prompted us to consider the instructors' perspectives on degrees of complexity of the multiple-choice questions, as well as the novices' responses, which determine the difficulty involved.

The remainder of the paper is structured as follows. In Section 2 we discuss relevant research identifying other approaches to understanding learning difficulties experienced by novices and the study of programming assessments. Section 3 presents our methodology, describing details of the exam questions and the data we seek to analyse. Results and discussion in the context of the Bloom's and SOLO learning taxonomies are presented in Section 4. Finally, we present our conclusions and proposed future endeavours in Section 5.

2 Related work

Research into the challenges of teaching programming to novices has long received attention, in several cases to better understand first year programming students' difficulties in conquering introductory programming concepts.

Kopec et al. (2007), analysed programmers' examination errors, but focussed on intermediate programmers. Intermediate programmers are those who have some programming experience and understand basic concepts of programming. Incorrect planning leads to incorrect answers. They concluded that educators must be very careful in their problem description and presentation, and that novices are often confused with nested loops and recursion, which differentiate intermediate to novice programmers' errors. In our study we focus on first year programming students, whom we define as novices. We are primarily interested in ensuring that novices have a good foundation to build upon in programming, to ensure that further study involving programming is well-received. Improved understanding of novice errors will also better inform educators about alleviating the difficulties experienced by novices at commencement.

A more recent study, Mike Lopez (2008) studied the responses of novices in an examination and found strong support for association between their code tracing and code writing skills, and between explaining code and writing code skills. They showed that there are strong correlations between code tracing and code writing. We also study the novices' responses to exam questions, however, our goal is to classify the questions and responses according to established taxonomies.

In order to consistently analyse answers to multiple-choice questions, one approach is to utilise learning taxonomies. Such taxonomies allow learning objectives to be classified according to cognitive domains. Benjamin Bloom created a taxonomy of thinking levels in the 1950's, known widely as *Bloom's Taxonomy* (Bloom 1956). Educators use Bloom's Taxonomy to guide the composition of challenging and sophisticated activities for learning. The cognitive domain of their study remains relevant today and is employed in identifying the educational levels required for course outcomes. According to Bloom's Taxonomy, there are three domains of the learning outcome, referred as; Affective, Psychomotor, and Cognitive. Our study focus on the Cognitive domain, which consist of six thinking levels: *knowledge*, *comprehension*, *application*, *analysis*, *synthesis* and *evaluation*.

In 2001, Anderson et al. (Anderson et al. 2001) revised the major categories in Bloom's Taxonomy to suit the emerging educational institutional needs of the new century. The revised Bloom's Taxonomy maintained the original ideas of Bloom, being the levels of cognition, but made changes within the categories, expanding them and explaining them better in the context of general education. In our study, we use the original Bloom's Taxonomy as we are classifying programming ability, and find that it is capable of explaining the ability or skills required by novices to answer each multiple-choice questions. However, although we have applied the original Bloom's Taxonomy , we recognise there are further explanations of cognitive load given in the revised version which would also apply to our novices' responses, but this other dimension can be employed at a later stage of course planning or renewing, instructional delivery and writing new assessment items.

It can be difficult to distinguish between the levels and categories when applying the taxonomy to each exam question after it has been written and appeared in the summative assessment (Thompson et al. 2008). This is due to the nature of programming exam papers, since there are rarely any keywords from Bloom's Taxonomy employed by the examiner for the questions. However, we have attempted to understand each question and explain our application of which particular level for each of the questions in the next section.

A number of studies have applied Bloom's Taxonomy to programming tasks. Scott (2003) explained some links between programming questions and Bloom's Taxonomy when he demonstrated how the taxonomy works in programming tests and provided some sample questions for each category of the taxonomy. Oliver et al. (2004) claimed that their Programming 1 course rated as 3.9 applying Bloom's Taxonomy, but, in their analysis, only assignments were rated according to the Bloom's Taxonomy. There was no evaluation made of examination questions based on Bloom's Taxonomy as due to the course designed, there is no final examination for Programming. Lister & Leaney (2003) identified the weak, middle and strong programming students in their study based on criterion-referenced grading (grades which were assigned according to criteria, irrespective of the resultant grade distribution). Different treatments, depending on the level within the taxonomy, were applied in order to obtain the various different grades. They proposed a scale based on the students' performances to determine their progression to the following semester. Furthermore, the study concluded that multiple-choice questions should not be seen as being too easy in the exam, since one third to one half of the class failed to achieve the 70% pass figure on their first attempt. However, they still believe that multiple-choice questions can provide a solid test of a student's knowledge and comprehension (Lister & Leaney 2003).

PeerWise (Denny et al. 2008) highlighted that there is a significant correlation between students' overall performances in exams and their contributions to the multiple-choice questions to the system. In our study, we too recognise the high value of multiple-choice questions, however, we focus on classifications via Bloom's Taxonomy of cognitive skills; we apply the guidelines of the cognitive process of the taxonomy to suit the questions written for the final exam.

Another taxonomy, the SOLO taxonomy (Biggs & Collis 1982), represents a more qualitative way to classify cognitive processes. There are five categories of SOLO taxonomy: *prestructural*, *unistructural*, *multistructural*, *relational*, *extended abstract* (Refer to Table 2). However, as originally specified (Biggs & Collis 1982), the SOLO categories are based on the age of the learner, such as the lowest category for youngest learner. We argue that while this claim has merit, learning is a process and there will always be something new to acquire, regardless of age, and hence the levels of cognition can be applied to learning as a process, regardless of age. There are also several studies using the SOLO Taxonomy that evaluate the responses of novices learning to read programs (Thompson 2007, Whalley et al. 2006, Lister et al. 2006). Lister et al. (2006) studied written and think-aloud responses from respondents, and reported that novices tend to understand code via a line-by-line approach, rather than understanding the code as a whole.

Our research question for this study is: What are the suitable taxonomies to classify the questions and responses in the final examinations of novice programmers?

3 Research methodology

Programming 1, is a core, first-year subject in every undergraduate degree, hence providing a high volume of representative data. It represents a first course in Java programming, with the aim of getting students to code simple, small Java programs involving, at their most challenging stage, a moderate-sized system with a range of classes (and simple inheritance requirements). We posit that novices give their best effort during their final examinations, in order to pass the subject, and such efforts are devoid of any collaboration with others, other than pre-exam revision with peers, and via consultation with teaching staff. Hence, errors in solutions presented in examinations are far more likely to be revealing of individual programming difficulties.

For our study we used as our data source the answers to selected questions in the semester 1, 2008 final examination paper for the subject Programming 1. The structure of the examination paper was as follows:

Part 1, worth a total of 30 (out of 100) marks, contained 20 equally-weighted short answer questions, of which 19 were multiple-choice questions and one was a code-tracing question. The 20 questions are attached in Appendix A.

Part 2 (worth 35 marks) contained 7 equally-weighted questions requiring code writing for short programs, or parts of programs(Refer Appendix B).

Finally, Part 3 required incremental development of code for a single problem context, which tested student application of simple object-oriented code completion or development from scratch. We provide no further details here as this part of the examination was beyond the the scope of the study reported here.

The time allocated to complete the paper was 3 hours plus an extra 15 minutes of reading time. On this basis the expected, estimated time to answer each multiple choice questions was approximately two to three minutes. There were no "explain in plain English" type of questions. However, sections of Part 2 required the recognition of code segments to be altered or extended.

Our focus was on the multiple-choice questions of Part 1 (primarily) and a selected question from Part 2, of the exam paper. In total there were 220 submissions received from the novices. We analysed the questions and answers to 19 of the 20 questions in Part 1. In each multiple-choice questions there is only one correct answer and three incorrect answers, denoted as *distracters*. We ignored question 6 because it was a code tracing question with no distracters. Here novices were required to trace a code segment and provide a single answer in the blank space provided, and not to select the correct answer from among a list of choices. The remaining 19 questions were "code interpretation" or "code reading" questions.

The exam paper was prepared by an experienced team of instructors (including one of the authors) in the second half of the semester, during which time the novices were enrolled in the subject. This allowed the instructors to adapt the paper to the current context of subject delivery, as needed. During the semester a range of preparatory instruments and activities were used to prepare novices to become more conversant with the style and complexity of the final examination paper. These included tutorial and laboratory exercises; a mid-semester test in Week 8, to test material covered up to and including Week 6 (the halfway point in a 12-week semester); an ongoing series of *Weblearn* quizzes and tests[1]; and discussions in lectures of model questions and answers from past exam papers, comparable to the style and complexity of what might be expected in their final exam.

In the remainder of this section we present descriptions of our approaches to classify the novice responses to the 19 multiple-choice questions (from Part 1 of the exam paper) according to Bloom's Taxonomy, and the responses to selected Part 2 questions according to the SOLO Taxonomy. The latter classification represents an extension of our study to include short-answer questions.

Our brainstorming to carry out such classifications was inconclusive, so we employed further analysis by introducing measures of instructor perceptions of question complexity and novice levels of difficulty. We present further discussion and analyses of these issues with the results in the next section.

[1]Weblearn is a School-developed online system for the management of subject question banks.

3.1 Methodology for classification of multiple-choice questions responses according to Bloom's Taxonomy

As discussed in Section 2, Bloom's Taxonomy was developed for describing and categorising the level of cognitive difficulty involved in learning a particular subject. We outline the Bloom's Taxonomy upon which we have based our categorisation of the multiple-choice exam questions in Table 1. In terms of cognitive complexity Knowledge is the lowest level category and relates to memorising information and being able to recall definitions. As the scale of complexity moves up, the cognitive factor increases, meaning that greater use is being made of the novice's mental capabilities. Evaluation is the highest level of cognition and relates to the creating, developing and writing of ideas and abstractions. The Application level 3 is the one where we believe most of the programming code questions from our exam paper have been pitched.

Table 1: Bloom's Taxonomy

Level	Category	Description
6	Evaluation	Test on the ability to evaluate ideas
5	Synthesis	Test on the ability to relate knowledge from several areas and use of old ideas to create a new one
4	Analysis	Test on the ability to understand the information and translate it into a different context
3	Application	Test on the ability to apply the information in a concrete situation, questions should be resolved using skills and knowledge
2	Comprehension	Test on the ability to understand the information and translate it into a different context
1	Knowledge	Test on the observation and recall of information learnt

3.2 Methodology for categorisation of code writing question for SOLO Taxonomy

We extended our study to analyse Question 24, a short answer question in Part 2 of the exam sheet, presented in Appendix B. We chose Question 24 as it requires short and precise responses and it provides clear instruction for the novices. It require the novices to write code to calculate the highest and lowest integer, from a set of integers passed via the commandline. As discussed earlier, the SOLO Taxonomy may be used to classify program code. Thus, we applied the SOLO Taxonomy to the novice answers. Our modified set of SOLO Taxonomy categories is presented in Table 2, where we have added the last category. We have slightly modified the categories to enable us to categorise all the responses.

According to SOLO Taxonomy, each level represents increasing cognitive load. The lowest level, Prestructural, reflects the situation when novices may have several pieces of unconnected information, but cannot make sense of the whole. However, as the levels scale up, in the extended abstract level, they demonstrate an ability to make the necessary connections between the subject area and generalising beyond the subject area to the world at large.

Table 2: Category and Descriptions of SOLO Taxonomy

Category	Descriptions
Extended Abstract	Novices able to make connections beyond the scope of question and able to transfer knowledge a new situation
Relational	Fully correct or almost right. Novices appreciate significance in relation to the whole program and can generalise outside of program
Multistructural	There are numbers of connections made. Novices can create code for loops and comparisons, but there are a few minor slips, leading to failure to connect the whole idea. They may fail to convert arguments, use incorrect operators, not interpret general explanation
Unistructural	Simple connections are made. Novices can compare, or write loops but fail to implement or derive the connections of loops in relation to manipulation of arrays or usage of further structures
Prestructural	There are bits of unconnected information. Novices know something, but the overall argument makes no sense
No attempt or totally wrong	The answer is blank or totally wrong

When classifying according to the SOLO Taxonomy, we investigate a particular solution as one whole model and then consider how the metacognitions may have formed. Thus, we found that it is important to analyse how each component part of the solution has been coded to build the whole relational model solution. We studied the interrelation of each components, in order to understand whether the novice could link all the components of the knowledge that they have learnt.

For example, below we outline a few components that should contribute to the relational model, in answering this question. The components are:

- Ability to create a loop
- Ability to extract or convert the argument correctly
- Ability to find the highest value
- Ability to find the lowest value
- Ability to code correctly

We used this approach to distinguish between Multistructural, Unistructural and Prestructural novice categories. We felt that the Extended Abstract level could not be tested in this question, as the novices had been give clear instructions to what question to solve.

4 Results and discussion

In this section we discuss primarily the classification of 19 multiple-choice questions (see Appendix A), by applying Bloom's Taxonomy, and analyse this with consideration of instructor assignment of level of question complexity, and novice measure of level of question difficulty.

We also applied the SOLO Taxonomy to novice responses to Question 24, a short, code-writing question (see Appendix B). For the novice to attain the Relational classification level of the SOLO Taxonomy, they must be able to connect the components which we identified in the previous section.

4.1 Application of Bloom's Taxonomy to Multiple Choice Questions

We examined the Multiple Choice Questions, their solutions and distracters, classified them according to Bloom's Taxonomy outlined in the previous section, and present these results in Table 3. We found that the content of the multiple-choice questions may be classified primarily into the three lower levels of Bloom's Taxonomy: Knowledge, Comprehension and Application, only. Since this is an introductory programming course, we would expect that the test instrument should test performance at the lower level skills. Earlier we had considered that the majority of the multiple-choice questions would rate in the Application level from the novices' viewpoint, however, it turned out that the majority of questions, 15 of them, are in the Comprehension level. We had difficulty distinguishing between categorising at the Comprehension and Application levels in the early stages of learning, and so we decided that, since this is a practical course, novices are expected both to comprehend and to be able to demonstrate the basic knowledge covered at the same time. There are three Application level questions in the examination, and these questions require the novices to display their ability to tackle an unfamiliar situation as posed in the question. There is only one question in the test which we categorised at the Knowledge or easiest level.

Table 3: Bloom's Taxonomy and the number of Multiple Choice Questions in each level

Level	Category	Number of Multiple Choice Questions
6	Evaluate	0
5	Synthesis	0
4	Analysis	0
3	Application	3
2	Comprehension	15
1	Knowledge	1

We realized that there are many sub-categories for each category presented by the taxonomy, as learning and background information are wide and varied. As such, we found that categorising a novice response was, possibly unsurprisingly, a difficult task which posed some interesting challenges. We also went on to find that it was difficult to categorise the question. Firstly the Bloom's taxonomy categories were not developed to relate to programming questions, and which "keyword" in each category to apply to the question at hand was not obvious.

In particular, we found it difficult to differentiate between categorising the questions according to the Comprehension and the Application levels. The distinction between Comprehension and Application levels is drawn between a question at the Comprehension level requiring a novice to abstract well enough to demonstrate the expected knowledge. Whereas at the Application level, when unfamiliar elements are presented, the novice should be familiar enough with the old existing elements to be able to restructure them and respond correctly to the question (Bloom 1956).

As an example, Question 9 and 15 (both in Section 7.1), both required novices to demonstrate their ability to iterate through loop(s). Question 9 is a 'do-while' loop, given the m value is 0, and novices are required to count how many time(s) the loop will iterate. This question is categorised as Comprehension level in Bloom's Taxonomy as the question required a novice to demonstrate the understanding that a 'do-while' must be executed atleast once. On the other hand, Question 15 required a novice to understand how the 'for-loop' will iterate(to demonstrate and understand of old existing elements) and present this as a matrix of 3x3. The method of presenting the data is the new elements. Thus, in order to solve the question, a novice needs to have the basic element (to understand the for-loop) and able to apply the knowledge and present the answer in a matrix.

4.1.1 Instructor's Level of Complexity

As a result of the challenges posed by categorisation of questions according to Bloom's Taxonomy, we pressed to investigate the motives of the instructors in setting the questions. We asked the instructors responsible for setting the exam to provide their assessment of the level of complexity for each question. The results of their assessments are presented in Table 6.

We based our study of the Instructor's Level of Complexity on the depth of the problem posed in the question. We suggested three basic categories to distinguish the complexity of the questions: Low, Medium and High. A Low complexity represents a very basic coverage of a particular concept, data element or instruction; a Medium level is when more than one action is involved, but the structure is sequential; and High refers to a question which may involve complex objects or nested structures or combinations of difficult concepts, such as if-statements inside loops. The instructor applied these categories as a basis for judging each question and we present these results in Table 4, the Instructor's Level of Complexity.

Table 4: Instructor Level of Complexity

Category	Description
Low	Definitions, variables, concepts, simple instructions
Medium	More than one action, object, if and loop statements
High	Nested structures, complex objects, combinations of difficult concepts

An associated question to consider was whether the instructor expectation of complexity was accurate. Instructor expectation as evidenced by the difficulty of the question may not be on a par with the novices' performances (Clear et al. 2008, Joni et al. 1983). We found it worthwhile and interesting not only to look at the complexity level expected by the instructor, but also how it compared with the responses given by the novices, or the Novice's Level of Difficulty.

4.1.2 Novice's Level of Difficulty

In our continuing analysis we looked at the level of difficulty faced by the novices. According to the nature of multiple-choice questions, they have four possible responses (A, B, C or D), only one correct answer and three distracters. Our scale for describing the novice difficulty is based on Lord (Lord 1952), and presented in Table 5. If there are 85% or more of the novices selecting the correct response, the question is an easy one. If only 51% to 84% of the novices pick the correct response, this question is of medium difficulty; and if 50% or less select the correct answer, the question is hard for the novices to resolve. Thus the novice level of difficulty of a question rates it as Easy, Medium or High.

Table 5: Novice Level of Difficulty

Level of Difficulty	Range of correct responses
Easy	85 - 100
Medium	51 - 84
Hard	0 - 50

According to these scales, there are 5 easy multiple choice questions, 10 medium ones and 4 hard questions on our exam paper. For the multiple choice questions, the summary of the content levels of Bloom's Taxonomy, levels of instructor complexity and levels of novice response difficulty for each question are presented in Table 6. Question 6 is omitted from this table, as it was not a multiple choice question.

Table 6: Question number (Q No), Bloom's Taxonomy (BT), Instructor Level of Complexity (Complexity) and Novice Level of Difficulty (Difficulty)

Q No	BT	Complexity	Difficulty
1	Knowledge	Low	Easy
2	Comprehension	Low	Medium
3	Comprehension	Low	Medium
4	Comprehension	Low	Medium
5	Comprehension	Low	Hard
7	Application	Low	Easy
8	Comprehension	Low	Easy
9	Comprehension	Low	Medium
10	Comprehension	Low	Hard
11	Comprehension	Medium	Hard
12	Comprehension	Medium	Medium
13	Comprehension	Medium	Medium
14	Comprehension	Medium	Easy
15	Application	High	Medium
16	Comprehension	High	Hard
17	Comprehension	Medium	Easy
18	Comprehension	Medium	Medium
19	Comprehension	Medium	Medium
20	Application	High	Medium

4.2 Application of SOLO Taxonomy to Short Answer Question

We applied the SOLO Taxonomy to the responses for question 24, which required novices to trace through a code segment about an array to determine the highest and the lowest integer values.

We found that 35.6% of the novices' answers were at the Relational category, 23.3% of responses were at the Multistructural level and less than 25% were at the lower level categories of the SOLO taxonomy.

4.3 Discussion

We started analysing the questions by using the Bloom's Taxonomy. We found that it is hard to distinguish between the Application and Comprehension

Table 7: SOLO Taxonomy applied to responses for the short written code segment

Category	Response
Relational	35.6
Multistructural	23.3
Unistructural	16.0
Prestructural	7.8
No attempt or totally wrong	17.4

level. At first, we thought that any question which has written code was devised to test the novices' ability at the Application level of Bloom's Taxonomy. We decided that the Comprehension and Application levels would be distinguished by the amount of abstraction required. If a novice needs to abstract in order to solve a question, then the question is developed to test at the Application level of Bloom's Taxonomy, whereas, if a novice does not need to abstract, then the question is developed to test at the Comprehension level of Bloom's Taxonomy.

We also found that, the three highest categories, Analysis (Level 4), Synthesis (Level 5) and Evaluation (Level 6), were not tested by this set of multiple choice questions for novice programmers. These three levels were tested and seen in the programming project conducted during the semester, whereby novices were able to practise their generating, planning and production skills, and also further evaluate other projects or code compared to their own.

We further analysed the Instructor's Level of Complexity as we wanted to understand their expectations when they devised the questions. We provide a simple measure as guideline for them to categorise the questions. We understand that the instructors may have strong background knowledge of programming, and thus tend to find the questions less complex than novices do.

There were a few tricky questions presented in this set of multiple-choice questions. This sort of question can be justified on the grounds of testing the novices with code which could well have been written by other novices and they need to know how to deal with such code and learn from it.

We decided it would be interesting to investigate any correlations between the categorisations of the novice responses, and the categorizations given to the questions. We stored the data in SPSS Package 15.0 and calculated the correlation between Bloom's Taxonomy and the Instructor's Level of Complexity. Next we looked for any correlation between Bloom's Taxonomy and the Novice's Level of Difficulty and finally between the Instructor's Level of Complexity and Novice's Level of Difficulty. The results are presented as in Table 8.

Table 8: Correlation(ρ) between pairwise Bloom's Taxonomy, Instructor Level of Complexity(Complexity) and Novice Level of Difficulty(Difficulty)

Variable	P-value
ρ Bloom's Taxonomy and Complexity	0.070
ρ Bloom's Taxonomy and Difficulty	0.941
ρ Complexity and Difficulty	0.468

Since the p-value for the correlation between Bloom's Taxonomy and the Instructor Level of Complexity is 0.070, which is greater than 0.05, we can say that there is no significant correlation between Bloom's Taxonomy and Instructor's level of Complexity at the confidence level of $\alpha = 0.05$. Similarly for the correlation between Bloom's Taxonomy and the Novice Level of Difficulty the p-value of 0.941 is greater than 0.05. Also for the correlation between the Instructor Level of Complexity and the Novice Level of Difficulty, the p-value is 0.468 which is much greater than 0.05. Hence we can say that there are no significant correlations between any of the taxonomies, Bloom's Taxonomy, the Instructor Level of Complexity, and the Novice Level of Difficulty, which means that all three are independent taxonomies, not related to each other at all.

In analysing the responses to the short answer question, we realised that the emphasis of the question was on the correct grammatical syntax, and no marks were awarded for having the logic or necessary problem-solving skills. For this example, the strong syntax knowledge requirement is like the pillar. If the novices are unable to practise or write their answer using the correct syntax, they may be discouraged from going further in answering that particular question. We concur with other studies which have employed the SOLO Taxonomy, that it is useful to evaluate the responses of novices learning to read programs (Thompson 2007, Whalley et al. 2006, Lister et al. 2006), but we also consider it is useful to measure the syntax knowledge as well.

5 Conclusions

We have found that it is very difficult to classify questions on the final exam paper using Bloom's Taxonomy It is hard to distinguish between the categories as the original Bloom's Taxonomy was written to suit the education field generally. Programming subjects are quite different in that the questions we create are not based on the keywords used in Bloom's Taxonomy, but are mostly based on snippets of code.

Since this is for an introductory programming course, the questions on our exam paper belonged to the lower, easier levels of Bloom's Taxonomy, but even then the questions posed at the Knowledge and Comprehension levels can be confusing and difficult for novices to answer. Although the content may be an if-statement, for instance, we found that depending on the nested alternatives, or the complexity of the test condition, that this can add other dimensions of difficulty to the question. Hence we incorporated further dimensions to our taxonomy to explain the Instructor Level of Complexity posed by the question, and the Novice Level of Difficulty relating to how they answered the question. We found the combination of these three dimensions allowed us to more clearly classify the questions.

Hence for classifying multiple-choice questions on exam papers, we recommend a three-dimensional taxonomy consisting of Bloom's categories for the content, the instructor's estimate of complexity, and the novice's percentage of correct responses as a measure of difficulty. We recommended the two additional measures as we have seen that there exist questions which are low level in complexity as determined by the instructors, but the novices found them very difficult to solve.

For classifying the code-writing questions, we recommend the SOLO Taxonomy, to measure the novices' understanding of the particular concepts tested. The SOLO Taxonomy provides a means of evaluating cognitive or mental models, to see if the novices are able to make connections between what they have learnt, if any exist.

Our future work will involve a further investigation to bridge the gap between the instructors and novices, in order to better understand the instructor's expectations in terms of complexity and the novices' difficulties. Our next goal is to minimise this gap so that both parties can work along together and overcome some of the difficulties faced by the novices.

6 Acknowledgment

The authors would like to thank Raymond Lister for his insight into Bloom's Taxonomy.

References

Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Bloom, B. S., Cruikshank, K. A., Pintrich, P. R. & Mayer, R. E. (2001), *A taxonomy for learning, teaching and assessing*, complete edition edn, Addison Wesley Longman, Inc, pp. 67–68.

Bennedsen, J. & Caspersen, M. E. (2007), 'Failure rates in introductory programming', *SIGCSE Bull.* **39**(2), 32–36.

Biggs, J. B. & Collis, K. F. (1982), *Evaluating the quality of learning. The SOLO Taxonomy (Structure of the Observed Learning Outcome)*, complete edition edn, Academic Press, pp. 61–92.

Bloom, B. S. (1956), *Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain*, New York: David McKay Co Inc., chapter Part II, pp. 62–197.

Bonar, J. & Soloway, E. (1983), Uncovering principles of novice programming, *in* 'POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages', ACM, New York, NY, USA, pp. 10–13.

Caspersen, M. E. & Bennedsen, J. (2007), Instructional design of a programming course: a learning theoretic approach, *in* 'ICER '07: Proceedings of the third international workshop on Computing education research', ACM, New York, NY, USA, pp. 111–122.

Clear, T., Edwards, J., Lister, R., Simon, B., Thompson, E. & Whalley, J. (2008), The teaching of novice computer programmers: bringing the scholarly-research approach to australia, *in* 'ACE '08: Proceedings of the tenth conference on Australasian computing education', Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 63–68.

Denny, P., Hamer, J., Luxton-Reilly, A. & Purchase, H. (2008), Peerwise: students sharing their multiple choice questions, *in* 'ICER '08: Proceeding of the fourth international workshop on Computing education research', ACM, New York, NY, USA, pp. 51–58.

Joni, S.-N., Soloway, E., Goldman, R. & Ehrlich, K. (1983), 'Just so stories: how the program got that bug', *SIGCUE Outlook* **17**(4), 13–26.

Kopec, D., Yarmish, G. & Cheung, P. (2007), 'A description and study of intermediate student programmer errors', *SIGCSE Bulletin* **39**(2), 146–156.

Lister, R. & Leaney, J. (2003), 'Introductory programming, criterion-referencing, and bloom', *SIGCSE Bull.* **35**(1), 143–147.

Lister, R., Simon, B., Thompson, E., Whalley, J. L. & Prasad, C. (2006), 'Not seeing the forest for the trees: novice programmers and the solo taxonomy', *SIGCSE Bull.* **38**(3), 118–122.

Lord, F. M. (1952), The relation of the reliability of multiple-choice tests to the distribution of item difficulties, *in* 'Psychometrika', SpringerLink, pp. 181–194.

Mike Lopez, Jacqueline Whalley Phil Robbins, R. L. (2008), Relationships between reading, tracing and writing skills in introductory programming, *in* 'ICER'08'.

Oliver, D., Dobele, T., Greber, M. & Roberts, T. (2004), This course has a bloom rating of 3.9, *in* 'ACE '04: Proceedings of the sixth conference on Australasian computing education', Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 227–231.

Oman, P. W., Cook, C. R. & Nanja, M. (1989), 'Effects of programming experience in debugging semantic errors', *J. Syst. Softw.* **9**(3), 197–207.

Scott, T. (2003), 'Bloom's taxonomy applied to testing in computer science classes', *J. Comput. Small Coll.* **19**(1), 267–274.

Smith, D. C., Cypher, A. & Tesler, L. (2000), 'Programming by example: novice programming comes of age', *Commun. ACM* **43**(3), 75–81.

Thomas, L., Ratcliffe, M. & Thomasson, B. (2004), 'Scaffolding with object diagrams in first year programming classes: some unexpected results', *SIGCSE Bull.* **36**(1), 250–254.

Thompson, E. (2007), Holistic assessment criteria: applying solo to programming projects, *in* 'ACE '07: Proceedings of the ninth Australasian conference on Computing education', Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 155–162.

Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M. & Robbins, P. (2008), Bloom's taxonomy for cs assessment, *in* 'ACE '08: Proceedings of the tenth conference on Australasian computing education', Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 155–161.

Wayne D. Gray, Neal. C. Goldberg, S. A. B. (1993), 'Novices and programming: Merely a difficult subject (why?) or a means to mastering metacognitive skills? (review of the book studying the novice programmer)', **9**(1), 131–140.

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A. & Prasad, C. (2006), An australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies, *in* 'ACE '06: Proceedings of the 8th Austalian conference on Computing education', Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 243–252.

7 Appendices

7.1 Appendix A

This appendix presents the text of selected questions from the exam paper, discussed and analysed in Section 4.

1. Which of the following is not a primitive type in Java ?
A) boolean
B) byte
C) String
D) float

2. What will be the value assigned to the variable x as a result of the following statement?

```
int x = 2 / 10 + 12 \% 4 + 5;
```

A) 5
B) 7
C) 8
D) 10

3. Which one of the given sets of X, Y and Z values makes the following expression true?

```
!( X && Y ) && ( !Y || Z )
```

A) X = false, Y = false, Z = false
B) X = false, Y = true, Z = false
C) X = true, Y = true, Z = false
D) X = true, Y = true, Z = true

4. Given that x and y are both variables of type int, the statements:
y = x + x + x;
y += y + x;
are equivalent to:
A) y = 4 * x;
B) y = 5 * x;
C) y = 7 * x;
D) y = 8 * x;

5. What will be assigned to the variable result after execution of the statements below, if the value that is entered by the user is -1?

```
int something = console.nextInt();
String result;
if (something <= 0)
   result = "ONE";
else if (something <= 50)
   result = "TWO";
if (something <= 100)
   result = "THREE"
else
   result = "FOUR";
```

A) "ONE"
B) "TWO"
C) "THREE"
D) "FOUR"

6. What is the output of the following code segment?

```
for (int i = 1; i <= 15; i += 2)
{
   if ( i % 3  ==  0)
      System.out.print("a");
   else
      System.out.print("b");
}
```

Answer:________________________

7. What is the output of the code segment below?

```
String message = "Sally is counting: ";
String numbers = "";
for (int i = 0; i < 12; i += 2)
   numbers = numbers + i + " ";
System.out.println(message + numbers);
```

A) Sally is counting: 0 2 4 6 8 10
B) Sally is counting: 0 1 2 3 4 5 6 7 8 9 10 11
C) Sally is counting: 0 2 4 6 8 10 12
D) Sally is counting: 0 1 2 3 4 5 6 7 8 9 10 11 12

8. What will be the output of the program segment below if marks is input as 50?

```
System.out.print("Enter marks : ");
int marks = console.nextInt();
if ( marks < 50 )
   System.out.print(" Fail");
if ( marks >= 50 )
   System.out.print(" Pass");
else if( marks >= 60 )
   System.out.print(" Credit");
if ( marks <= 70 )
   System.out.print(" Distinction");
else
   System.out.print(" High-Distinction");
```

A) Pass Credit
B) Pass Distinction
C) Pass High-Distinction
D) Pass Credit Distinction High-Distinction

9. How many times will the do-while loop below be executed?

```
int m = 0;
do {
   System.out.println(m);
   m = m - 1;
    } while (m > 0);
```

A) 0 times
B) 1 time
C) 10 times
D) It is an infinite loop (ie. it will never stop executing)

10. Which of the following statements is false?
A) A static method can be accessed directly via the class without having to create an object from that class first (visibility permitting)
B) A non-static (instance) method cannot refer to a static variable inside the same class
C) A static method cannot refer to non-static (instance) variable inside the same class
D) All objects created from a class which includes a static variable may change the value stored in that static variable (visibility permitting)

11. What are the values of the variables a and b after the execution of the following program:

```
public class TestParameterPassing
{
   public static void main (String [] args)
   {
      double a = 2.5;
      String b = "Hello";

      anyMethod(a, b);
   }
   public static void anyMethod (double d, String s)
   {
     d = d * 3;
     s = "Goodbye";
  }
}
```

A) a = 2.5, b = "Goodbye"
B) a = 2.5, b = "Hello"
C) a = 7.5, b = "Goodbye"
D) a = 7.5, b = "Hello"

12. Which one of the following statements is true?
A) An abstract class must have an abstract method
B) All methods in an abstract class must themselves be abstract
C) An abstract class cannot define instance variables
D) An abstract class cannot be instantiated

13. Which of the following statements is true in regards to files?
A) You cannot write numeric values to a text file
B) A text file that has been opened for writing must be closed after the program has finished writing data in order for the data to be written out correctly.
C) No exception is thrown if a text file that does not exist is

opened for reading
D) An exception is thrown if a text file that does not exist is opened for writing

14. Given that B is a subclass of A and C is a subclass of B, what is printed by the following code segment ?

```
A a = new C();
if (a instanceof A)
   System.out.print("yes1 ");

if (a instanceof B)
   System.out.print("yes2 ");

if (a instanceof C)
   System.out.print("yes3");
```

A) yes1
B) yes2
C) yes1 yes2
D) yes1 yes2 yes3

15. What is the output of the program below?

```
public static void main (String[] args)
{
   int[][] m = new int[3][3];

   for (int row = 0; row <= 2; row++)
      for (int col = 0; col <= 2; col++)
    m[row][col] = col;

   for (int i=0; i<3; i++)
   {
      for (int j=0; j<3; j++)
   System.out.print("   "+m[i][j]);
      System.out.println();
   }
}
```

```
A) 0 1 2
   0 1 2
   0 1 2

B) 0 0 0
   1 1 1
   2 2 2

C) 2 1 0
   2 1 0
   2 1 0

D) 2 2 2
   2 2 2
   2 2 2
```

16. What is the output produced by the following code segment (assuming that the exception types ExceptionTypeOne and ExceptionTypeTwo have been defined previously and are not related by inheritance)?

```
public void aMethod()
{
   int value = 10;
   try
   {
      System.out.print("one ");
      if (value <= 10)
      {
         System.out.print("two ");
         throw new ExceptionTypeTwo();
         System.out.print("three ");
      }
      catch (ExceptionTypeOne e)
      {
         System.out.print("four ");
      }
      finally
      {
         System.out.print("five ");
      }
      System.out.print("six ");
   }
}
```

A) one two
B) one two five
C) one two three five six
D) one two four five six

17. If the value of the variable x is -10, 10 and 100 respectively then what is printed after the execution of the following code segment for each of the values?

```
if (x > 50 && x > 0)
               System.out.println("Success!");
            else
               System.out.println("Failure!");
```

A) Failure!, Failure!, Success!
B) Failure!, Success!, Failure!
C) Failure!, Failure!, Failure!
D) Success!, Success!, Failure!

18. Consider the following code segment?

```
int[ ] x = {2, 1, 4, 5, 7};
int limit = 3;
int i = 0;
int sum = 0;

while ((sum < limit) && (i < x.length))
{
   ++i;
   sum += x[i];
}
```

What value is in the variable ”i” after this code is executed?
A) 0
B) 1
C) 2
D) 3

19. Consider the following code segment:

```
int[ ] x = {0, 1, 2, 3};
int temp;
int i = 0;
int j = x.length-1;

while (i < j)
{
temp = x[i];
x[i] = x[j];
x[j] = 2*temp;
i++;
j--;
}
```

After this code is executed, array ”x” contains the values:
A) 3, 2, 2, 0
B) 0, 2, 2, 3
C) 0, 2, 4, 6
D) 6, 4, 2, 0

20. Consider the following code segment.

```
int[ ] x1 = {1, 2, 4, 7};
int[ ] x2 = {1, 2, 5, 7};
int i1 = x1.length-1;
int i2 = x2.length-1;
int count = 0;
while ((i1 > 0) && (i2 > 0))
{
if (x1[i1] == x2[i2])
{
++count;
--i1;
--i2;
}
else if (x1[i1] < x2[i2])
{
--i2;
}
else
{ // x1[i1] > x2[i2]
--i1;
}
}
```

After the above while loop finishes, ”count” contains what value?
A) 3
B) 2
C) 1
D) 0

7.2 Appendix B

This is the short answer question that is analysed using the SOLO Taxonomy.

Question 24
Complete the HighLow class below to identify and display the highest and lowest of the series of positive integer values passed into the program as command line arguments.

Expected Input/Output is shown below.
java HighLow 7 4 9 10
Highest value passed in was 10
Lowest value passed in was 4

java HighLow 45 52 81 69 23 97 76
Highest value passed in was 97
Lowest value passed in was 23

Notes:
Command-line arguments are passed to the main method through the array of String references (args in the main method below). The size of any array can be accessed through its length attribute (note: you can assume at least one valid argument will be passed in on the command line). You will need to use Integer.parseInt() to convert each command line argument to integer format before processing it.
(5 marks)

```
public class HighLow
{
   public static void main (String[] args)
   {
      int highestArg = 0;
      int lowestArg = 0;
      int nextArg;
------------------------------------------
------------------------------------------
------------------------------------------

      System.out.println(
      "Highest value passed in was " +
      highest);
      System.out.println(
      "Lowest value passed in was " +
      lowest);
   }
}
```

Ten Years of the Australasian Computing Education Conference

Simon
University of Newcastle
Australia

simon@newcastle.edu.au

Abstract

The Australasian Computing Education Conference is now in its eleventh year. This paper charts the ups and downs of the conference from its origin in 1996, through its troubled years, to its recent apparently steady state. All 328 papers from the ten conferences are classified according to Simon's system for classifying computing education papers, and features of interest are pointed out. Only one clear trend over time is observed, and that is a steady and distinct increase in the proportion of research papers. The analysis then moves from the papers to their 496 distinct authors, exploring where the authors come from, how many papers each has contributed to the conference, and which authors appear to have made this conference their home. A final look at the number of papers presented each year suggests that the conference might once more be experiencing difficulty, and speculates on its future.

Keywords: classifying publications, computing education.

1 Introduction

The Australasian Computing Education Conference, formerly known as the Australasian Computer Science Education Conference, has been held ten times over the 13 years from 1996 to 2008. This paper gives an overview of the ten offerings, then goes on to briefly analyse the papers that have been presented at the conference and the authors of those papers.

2 Ten offerings in 13 years

Table 1 summarises the ten offerings of the conference. The remainder of this section describes the main points of interest during that time.

The first Australasian Computer Science Education Conference was held in Sydney in 1996. Chaired by John Rosenberg and Alan Fekete of Sydney University, it provided a regional forum for the presentation of work that might otherwise have been submitted to the SIGCSE Technical Symposium or ITiCSE. (SIGCSE is the Special Interest Group on Computer Science Education, a SIG of the premier computing professional group the ACM; the Technical Symposium is its annual conference in the US, and ITiCSE (Innovation and Technology in Computer Science Education) is its annual conference in Europe.)

Copyright © 2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

Australian academics appear to have leapt at the opportunity to submit their work closer to home than had previously been possible, but there were also papers from overseas, their authors perhaps taking advantage of a funded trip to a desirable destination. Of the 51 accepted papers, the authors of just over 70% were from Australia, with the remainder coming from the USA (6 papers), New Zealand (4 papers), the UK (4 papers), and Japan (1 paper).

At the end of the conference an interested group met to decide whether to continue. It was assumed that the first offering had attracted a backlog of papers that had built up over some years, and it wasn't clear whether the steady state would provide enough papers to warrant running the conference on an annual basis. Even so, it was agreed to try the following year and see what would happen.

The second conference did indeed attract rather fewer submissions, but still enough for the conference to run. At this point it was agreed that there did appear to be sufficient interest to support an annual conference.

Plans at this stage were somewhat ad hoc: towards the end of each conference a group of willing parties would meet and somebody would volunteer to host and chair the next conference. This arrangement failed in 1999, when the volunteers didn't manage to bring things together. This meant not only that there was no conference in 1999, but that there was no meeting to decide on chairs and a venue for the subsequent conference. Realising this, Judy Sheard and Dianne Hagan of Monash University in Melbourne took the initiative and ran the conference in 2000.

Two matters that had often been discussed at the meetings of interested parties were the conference name and the conference logistics. Some felt that the 'computer science' in the conference name was unnecessarily restrictive, effectively denying legitimacy to other areas of computing such as information systems. This feeling was acted on in 2000, when the name was changed to the Australasian Computing Education Conference.

The question with regard to logistics was whether ACSE (now ACE) should collocate with the Australasian Computer Science Conference. The latter conference, which had been running for more than 20 years, was making economies of scale by gathering a number of smaller conferences together. The potential benefit to ACE was that organisational matters such as registration, venue, catering, and proceedings would be taken care of by the organisers of Australasian Computer Science Week (ACSW) as the combined conference was known. The main potential cost was the move from mid-year to January, which was seen for various reasons as a less convenient time. The temptation to join ACSW was strong, but it was resisted.

Table 1: summary of the ten offerings

Conference	Location	Chairs	Submitted	Accepted	Accept rate
ACSE 1996	Sydney	John Rosenberg, Alan Fekete	114	51	45%
ACSE 1997	Melbourne	John Hurst, Harald Søndergaard	46	31	67%
ACSE 1998	Brisbane	Paul Strooper, David Carrington	59	27	46%
ACE 2000	Melbourne	Judy Sheard, Dianne Hagan	79	39	49%
ACE 2003	Adelaide	Tony Greening, Raymond Lister	47	34	72%
ACE 2004	Dunedin	Raymond Lister, Alison Young	87	48	55%
ACE 2005	Newcastle	Alison Young, Denise Tolhurst	67	32	48%
ACE 2006	Hobart	Denise Tolhurst, Samuel Mann	60	29	48%
ACE 2007	Ballarat	Samuel Mann, Simon	43	20	47%
ACE 2008	Wollongong	Simon, Margaret Hamilton	39	18	46%

After ACE 2000 there was another lost year, when ACE 2001 failed to eventuate. This time the rescue was performed by Tony Greening and Raymond Lister, who, believing that the logistical problems were part of the reason for the failures, decided unilaterally to combine with ACSW. This explains the other apparent missing year, as the next conference was held not in June or July 2002 but in January 2003.

Another important decision made at this time was to provide for continuity of chairs, so that each year there would be one chair who had already run an ACE. Raymond Lister began the process by chairing ACE 2004 as well as ACE 2003, and each subsequent chair has spent two years in the job, first as a junior chair learning what was involved, then as a senior chair showing the ropes to the new junior.

The high paper acceptance rates of ACSE 1997 and ACE 2003 can perhaps be ascribed in part to a desire to accept a reasonable number of papers despite the lower numbers of submissions in those years. Submission numbers fell again for ACE 2007 and ACE 2008, but by this time the chairs felt constrained to keep the acceptance rate below 50% for reasons of quality assurance, even though this meant a serious reduction in the number of papers presented.

3 The papers

In all, 328 papers have been accepted and presented at the ten offerings of the conference. In this section the papers are analysed, first according to Simon's system for classifying computing education papers (Simon 2007, Simon et al 2008), and subsequently with some additional facts and figures that might be of interest.

Simon's system classifies a computing education paper according to four dimensions: the context in which the work presented is set; the theme of the paper, what it is about; the scope of the work, which indicates the breadth of the context; and the nature of the paper, an indication of whether it is a research paper, an experience report, or a position paper or proposal. The dimensions will be explained further in the following subsections, illustrated with examples from ACE.

3.1 Context

A paper's context is typically the subject matter of the course or subject in which it is taught. Therefore we would expect to see papers with contexts such as

Table 2: contexts of the 328 papers

Context	Papers	Context	Papers
artificial intelligence	1%	intro to IT	2%
broad-based	23%	literature	1%
capstone project	5%	logic	1%
compilers	<1%	management	<1%
computer forensics	<1%	networks	4%
data structures	2%	operating systems	1%
database	2%	postgraduate / research	1%
design	<1%	programming languages	1%
eBusiness/eCommerce	<1%	programming	32%
ethics/professionalism	1%	school outreach	1%
formal methods	1%	software engineering	5%
graphics	<1%	system modelling	<1%
group work	2%	systems analysis	2%
hardware/architecture	4%	web use	1%
human-computer interface	2%	webpage development	1%
image processing	<1%	work experience	1%
information systems	3%		

programming (*"Uni cheats racket": a case study in plagiarism investigation* (Zobel 2004)), information systems (*Authentication strategies for online assessments* (Summons & Simon 1998)), compilers (*Jocula - an instructive compiler* (Buckley & Hext 1996)), and so on. In addition, the system recognises three contexts that do not represent curricular subjects. The *group work* context is used for papers that, regardless of the subject matter, concentrate on aspects of group management, dynamics, or assessment (*Developing the software engineering team* (Hogan & Thomas 2005)). The *literature* context is for papers, typically surveys, whose data comes from the literature rather than the classroom (*A citation analysis of the ACE2005 - 2007 proceedings, with reference to the June 2007 CORE conference and journal rankings* (Lister & Box 2008)). And *broad-based* is used for papers that have no identifiable context (*Building a rigorous research agenda into changes to teaching* (Daniels et al 1998)) and for papers that range across multiple contexts (*Attracting and retaining females in information technology courses* (Clayton et al 1996)).

The 328 papers together cover 33 contexts, as shown in table 2. Programming accounts for 32% of the papers, a further 23% are broad-based, and the remainder make up a broad and shallow spread over the remaining 31 contexts. The spread is reasonably uniform across the ten offerings, with no noticeable trends over time.

3.2 Theme

The theme of a paper is what the paper is actually about, and at first consideration might be confused with its context. *Language tug-of-war: industry demand and academic choice* (de Raadt et al 2003) might appear to be about programming, but that is in fact its context. The paper is about the teaching technique of which programming language to use, and so it fits into the theme of teaching/learning techniques. In a similar vein, *The case for more digital logic in computer architecture* (Hoffman 2004) has a context of hardware/architecture but a theme of curriculum, and *Self and peer assessment in software engineering projects* (Clark et al 2005) has a context of capstone projects but a theme of assessment tools, as it presents a tool developed by the authors to assist with the assessment process.

While the set of possible contexts is limited only by the set of papers being examined, the set of themes remains fairly fixed. The themes of the 10 years of ACE papers are shown in figure 1.

By far the bulk of the papers are about teaching/learning techniques (how we teach), teaching/learning tools (tools to help us teach), and curriculum (what we teach). Assessment techniques and tools together make up some 13% of the papers; students' ability and aptitude makes up about 5%, as does online and/or distance delivery; and the remaining contexts each account for less than 3% of the papers.

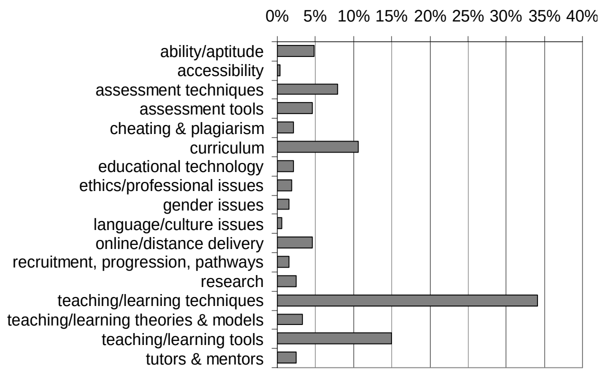


Figure 1: themes of the 328 papers

3.3 Scope

The scope of a paper is an attempt to specify the extent of collaboration with the (computing) education community that the work entailed. The narrowest recognised scope is the single subject (or course). A paper set in a single subject might possibly have been written with no collaboration at all (*Teaching software testing* (Carrington 1997)), although the many multi-author single-subject papers attest that this need not be the case (*Transforming learning of programming: a mentoring project* (D'Souza et al 2008)).

The program/department scope indicates a paper that is set in several distinct subjects across a degree program or a department. Such papers generally entail collaboration within the department (*Performance and progression of first year ICT students* (Sheard et al 2008)), although there are a handful of single-author program/department papers (*Peer mentoring female computing students - does it make a difference?* (Craig 1998)).

The scope of institution, recognising collaboration with colleagues in other departments at the same institution, tends to be quite rare. It is not always easy to break the silo mentality, but it can be rewarding to do so (*Peer assessment using Aropä* (Hamer et al 2007)).

It is generally easier to collaborate with computing education colleagues at other institutions, so there are many papers whose scope is many institutions (*eScience curricula at two Australian universities* (Gardner et al 2005)), especially since the advent of papers arising from working parties or workshops (*Differing ways that computing academics understand teaching* (Lister et al 2007)).

Some papers do not have an identifiable scope, typically because they have no explicit context (*Multiple choice questions not considered harmful* (Woodford & Bancroft 2005)) or because their context is the literature (*Qualitative research projects in computing education research: an overview* (Berglund et al 2006)). These papers are assigned a scope of not applicable.

There is no systematic variation in the pattern of scopes over time, so figure 2 shows the combined scopes of the papers from the last ten years.

3.4 Nature

The nature dimension was designed to acknowledge and chart the distinction between papers that are clearly reporting on research and papers that report their authors'

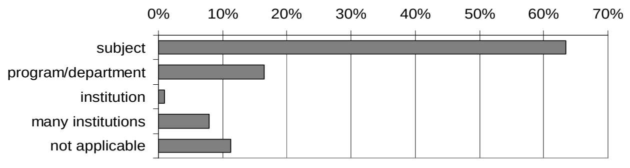


Figure 2: scopes of the 328 papers

experiences when implementing particular concepts in their classrooms. The intent is not to value individual research papers more highly than individual practice papers, but to recognise and applaud any overall increase in the amount of unequivocal research being reported in a body of papers.

An experiment paper (*The neglected battle fields of syntax errors* (Kummerfeld & Kay 2003)) reports on a scientific-style experiment, at the very least entailing a control group and an experimental group. It is logistically and ethically challenging to carry out such experiments in the classroom, with different groups being taught in different ways, so experiment papers tend to be rare in education.

A study paper reports on the implementation of a study designed to address a particular research question. The study will be carried out, data will be gathered and analysed, and conclusions will be drawn (*Mental models, consistency and programming aptitude* (Bornat et al 2008)).

An analysis paper is just as rigorous as an experiment or study paper, but addresses its research question by analysing existing data rather than first generating it. Analysis papers might be based on collected student results (*The impact on student performance of a change of language in successive introductory computer programming subjects* (Doube 2000)), on published literature (*A citation analysis of the ACE2005 - 2007 proceedings, with reference to the June 2007 CORE conference and journal rankings* (Lister & Box 2008)), or anywhere else where interesting data might already exist (*Decoding doodles: novice programmers and their annotations* (Whalley et al 2007)).

Report papers, the staple of computing education conferences, are the means by which academics exchange their experiences with (generally) new tools and techniques in the classroom. Valentine (2004) called publications of this type Marco Polo papers: ‘I went there and I did that’. Perhaps the term Genesis papers would be more fitting: ‘and he saw what he had done, and it was good’. Even where such a paper concludes by presenting the results of a student survey showing approval of the change, the survey result is incidental to the experience report, and does not shift the paper into the study or analysis categories.

Position/proposal papers outline work that is to yet be done, new ideas that are yet to be put into practice, or their authors’ thoughts on a particular question (*The case for more digital logic in computer architecture* (Hoffman 2004)).

With the five categories described above, it seems reasonable to classify experiment, study, and analysis papers as unequivocally research. They propose a research question, they gather the data to answer that question, they analyse the data, and infer the result. While some people would argue that reports, position papers, and proposals are also research, this is generally a lot less clear cut. There might indeed be some papers in those groups with a legitimate claim to be called research, but most of them are probably not.

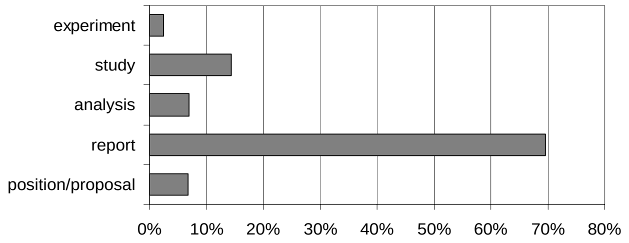


Figure 3: natures of the 328 papers

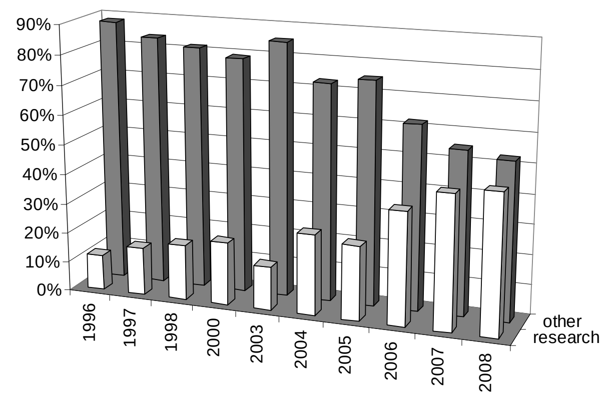


Figure 4: proportions over time of research papers (experiment, study, and analysis) and other papers (report, position/proposal)

Figure 3 shows the natures of the 328 papers from the ten offerings of ACE. This time, though, there is a clear trend over time. Simplifying the scale to research papers (experiment, study, and analysis) and other (report and position/proposal), figure 4 shows a steady growth in the proportion of research papers over the lifetime of the conference. This is a pleasing observation.

3.5 Titles

One cannot examine so many papers without noticing aspects of their titles. Some titles appear to be deliberately amusing or baffling; others are short and to the point; others appear to be trying to tell the whole story and save readers the effort of reading the paper.

Figure 5 shows the title lengths of the papers, from the single two-word title (*Why Ada?* (Millar & Mohammadian 1996)) to the single 25-word title (*One small step toward a culture of peer review and multi-institutional sharing of educational resources: a multiple choice exam for first semester programming students* (Lister 2005)). On examining figure 5, one wonders whether the chairs of ACE 2009 might look kindly on papers with 8-word titles, to help bring the overall distribution closer to normal.

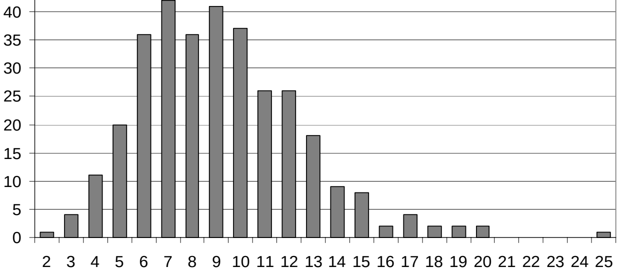


Figure 5: lengths of titles of the 328 papers

4 The authors

Over the ten years there have been 496 distinct authors of ACE papers. Many of those authors have only ever (co-) authored a single paper at the conference, while others have come back many times.

Table 3 shows the number of authors who have authored given numbers of papers, putting names to those who have contributed most. These repeat contributors are obviously the people one would expect to see at any ACE. All have their stories, of which a few are mentioned here.

Raymond Lister, the most prolific ACE author, is one of the two chairs who revived the conference in 2003, ran it in conjunction with ACSW, and brought in the two-year terms for junior and then senior chair.

Simon is the only author who has had a paper at every ACE since time began. Mats Daniels has had papers in 8, Raymond Lister and Angela Carbone in 7, and Ilona Box, Anders Berglund, and Judy Sheard in 6.

John Hamer is the highest-contributing author from New Zealand.

Mats Daniels is the highest-contributing author from outside Australasia, closely followed by Anders Berglund. Both are from Sweden.

Michael de Raadt is the highest-contributing author all of whose papers are in the research grouping of experiment, study, and analysis. He is closely followed by Anthony Robins.

Peter Bancroft is the highest-contributing author none of whose papers are in the research grouping.

Nicole Herbert/Clark is the highest-contributing author who is known to have changed her name during the lifetime of ACE. Tracking authors through a change of name requires inside knowledge, so there might be others beyond the three recognised in this analysis.

The average number of authors to a paper is 2.4. For most of the life of the conference it sat close to 2, but then a surge in multi-author papers drove it up to nearly 4 in 2006, after which it fell to 3.4 in 2007 and 3 in 2008.

The highest number of authors for a single paper was 21 (*Differing ways that computing academics understand teaching* (Lister et al 2007)), while the previous year saw three 15-author papers from a single project (Simon, Cutts et al 2006, Simon, Fincher et al 2006, Tolhurst et al 2006).

Table 3: authors contributing given numbers of papers

Number of papers	Achieved by number of authors
14	1 (Raymond Lister)
13	1 (Simon)
9	3 (Ilona Box, John Hamer, Margaret Hamilton)
8	1 (Judy Sheard)
7	7 (Angela Carbone, Mats Daniels, Michael de Raadt, Tony Greening, Patricia Haden, Judy Kay, Jodi Tutty)
6	4 (Peter Bancroft, Anders Berglund, Anthony Robins, Errol Thompson)
5	4 (Alan Fekete, Nicole Herbert/Clark, Marian Petre, Denise Tolhurst)
4	11
3	30
2	72
1	362

4.1 Where they're from

Analysis of where the papers come from will use the simplification that a papers comes from where its first author comes from. Figure 6 shows the proportions of papers from Australia, New Zealand, and other countries over the ten years.

For the first three offerings about 70% of the papers were from Australia, with reasonable proportions from New Zealand and other countries (Germany, Japan, Taiwan, Sweden, UK, and USA).

In the troubled years, 2000 and 2003, nearly all of the papers were from Australia, with just three from New Zealand, two each from UK and USA, and one each from Denmark, Germany, and Sweden. It would seem reasonable to conclude that the uncertainty surrounding the conference might have made overseas academics reluctant to submit papers to it, or perhaps even unaware that it was still running.

Once the conference was back on track the proportion of papers from New Zealand increased to a fairly steady 30%, and the proportion from other countries (Finland, Ireland, Norway, South Africa, Sweden, UK, and USA) has sat around 15%-20%. The 'Australasian' tag seems to be warranted, and the conference draws a good number of papers from a broad range of countries outside the region.

5 The future

Another look at table 1 shows that, while the numbers of accepted papers have been up and down over the years, the past two years have been among the lowest. Submissions are down, and the conference chairs no longer have the luxury of boosting numbers by accepting a greater proportion of the submitted papers – to do so would almost certainly result in a drop in the quality ranking of the conference within Australia and perhaps elsewhere.

There has been some speculation as to whether the non-metropolitan locations of ACSW 2007 and ACSW 2008 led to this downturn, in which case ACE 2009, in Wellington, New Zealand, should see numbers pick up again. Unfortunately, the recently announced figures for ACE 2009 show that there were exactly the same number of submissions and the same number of acceptances as for ACE 2008. One must wonder why the number of

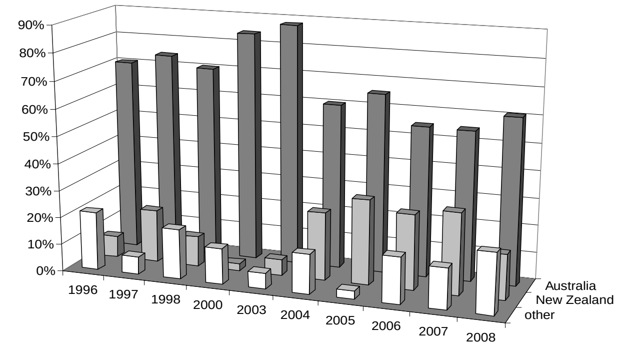


Figure 6: proportions of papers each year from Australia, New Zealand, and other countries

submissions has been so low since 2007.

Many universities and polytechnics in both Australia and New Zealand have recently made dramatic cuts to their academic staff numbers. This affects research in two ways: first, there are fewer people to conduct it; and second, those who do remain in academic work are expected to do more teaching, and thus have less time for research. This will clearly have a lasting impact on the overall output of research, and therefore on the number of papers submitted and accepted to conferences.

Another possible factor is the push in many countries for 'quality assurance', one consequence of which is that people are expected to publish more in high-ranking journals and high-ranking conferences. It seems all but impossible to have ACE recognised as a high-ranking conference, so institutions might increasingly be seen as discouraging their staff from submitting papers to it.

Finally, it is possible that the higher costs of travel and conference attendance are ruling it out as options for an increasing number of academics.

It would be nice to believe that the current drop in submissions and acceptances is short-lived, but we must accept the possibility that it is the beginning of the end for the conference.

6 Conclusion

The Australasian Computing Education Conference has been run ten times over the 13 years of its existence, surviving some difficult times in the process.

The 328 papers presented at the conference have been based predominantly in the context of programming or in no particular context, with a further 31 contexts each accounting for no more than 5% of the papers. The bulk of the papers deal with the themes of teaching/learning techniques, teaching/learning tools, and curriculum; a reasonable number deal with assessment techniques, ability/aptitude, assessment tools, and online/distance delivery; and the remainder are spread among ten further themes. More than 60% of the papers are set in single subjects, with about 10% in multiple subjects within the same department or degree program and about 10% set across two or more institutions. Nearly 70% of the papers are experience reports or 'Genesis papers', but the proportion of papers that are unequivocally research shows a steady increase from just over 10% in 1996 to nearly 50% in 2008.

The conference has seen papers by 496 distinct authors, of whom 362, nearly three-quarters, have had only one paper at ACE. At the other end of the scale, 21 authors have had five or more ACE papers, and the two most prolific have had 14 and 13 papers.

The bulk of the papers have always been from Australia, but recent years have seen respectable proportions of papers from New Zealand (about 30%) and nearly a dozen other countries (10%-15%).

Until recently the conference appeared to have good prospects for a long future. However, it does seem to have suffered a recent downturn in the numbers of papers submitted and accepted, for reasons that are not entirely clear, and it remains to be seen whether ACE can survive this difficult time as it has survived others in the past.

7 References

A Berglund, M Daniels, & A Pears (2006). Qualitative research projects in computing education research: an overview. Eighth Australasian Computing Education Conference, ACE 2006, Hobart, Australia.

R Bornat, S Dehnadi, & Simon (2008). Mental models, consistency and programming aptitude. Tenth Australasian Computing Education Conference, ACE 2008, Wollongong, Australia.

R Buckley & J Hext (1996). Jocula - an instructive compiler. First Australasian Computer Science Education Conference, ACSE 1996, Sydney, Australia.

D Carrington (1997). Teaching software testing. Second Australasian Computer Science Education Conference, ACSE 1997, Melbourne, Australia.

N Clark, P Davies, & R Skeers (2005). Self and peer assessment in software engineering projects. Seventh Australasian Computing Education Conference, ACE 2005, Newcastle, Australia.

D Clayton, M Cranston, & M Lynch (1996). Attracting and retaining females in information technology courses. First Australasian Computer Science Education Conference, ACSE 1996, Sydney, Australia.

A Craig (1998). Peer mentoring female computing students - does it make a difference? Third Australasian Computer Science Education Conference, ACSE 1998, Brisbane, Australia.

M Daniels, M Petre, & A Berglund (1998). Building a rigorous research agenda into changes to teaching. Third Australasian Computer Science Education Conference, ACSE 1998, Brisbane, Australia.

M de Raadt, R Watson, & M Toleman (2003). Language tug-of-war: industry demand and academic choice. Fifth Australasian Computing Education Conference, ACE 2003, Adelaide, Australia.

W Doube (2000). The impact on student performance of a change of language in successive introductory computer programming subjects. Fourth Australasian Computing Education Conference, ACE 2000, Melbourne, Australia.

D D'Souza, M Hamilton, J Harland, P Muir, C Thevathayan, & C Walker (2008). Transforming learning of programming: a mentoring project. Tenth Australasian Computing Education Conference, ACE 2008, Wollongong, Australia.

H Gardner, C Johnson, G Leach, & P Vuylsteker (2005). eScience curricula at two Australasian universities. Seventh Australasian Computing Education Conference, ACE 2005, Newcastle, Australia.

J Hamer, C Kell, & F Spence (2007). Peer assessment using Aropä. Ninth Australasian Computing Education Conference, ACE 2007, Ballarat, Australia.

ME Hoffman (2004). The case for more digital logic in computer architecture. Sixth Australasian Computing Education Conference, ACE 2004, Dunedin, New Zealand.

J Hogan & R Thomas (2005). Developing the software engineering team. Seventh Australasian Computing

Education Conference, ACE 2005, Newcastle, Australia.

SK Kummerfeld & J Kay (2003). The neglected battle fields of syntax errors. Fifth Australasian Computing Education Conference, ACE 2003, Adelaide, Australia.

R Lister (2005). One small step toward a culture of peer review and multi-institutional sharing of educational resources: a multiple choice exam for first semester programming students. Seventh Australasian Computing Education Conference, ACE 2005, Newcastle, Australia.

R Lister, A Berglund, I Box, C Cope, A Pears, C Avram, M Bower, A Carbone, B Davey, M de Raadt, B Doyle, S Fitzgerald, C Kutay, L Mannila, M Peltomäki, J Sheard, Simon, K Sutton, D Traynor, J Tutty, & A Venables (2007). Differing ways that computing academics understand teaching. Ninth Australasian Computing Education Conference, ACE 2007, Ballarat, Australia.

R Lister & I Box (2008). A citation analysis of the ACE2005 - 2007 proceedings, with reference to the June 2007 CORE conference and journal rankings. Tenth Australasian Computing Education Conference, ACE 2008, Wollongong, Australia.

JWL Millar & M Mohammadian (1996). Why Ada? First Australasian Computer Science Education Conference, ACSE 1996, Sydney, Australia.

J Sheard, A Carbone, S Markham, AJ Hurst, D Casey, & C Avram (2008). Performance and progression of first year ICT students. Tenth Australasian Computing Education Conference, ACE 2008, Wollongong, Australia.

Simon (2007). A classification of recent Australasian computing education publications. Computer Science Education 17, 3, 155-169.

Simon, Q Cutts, S Fincher, P Haden, A Robins, K Sutton, B Baker, I Box, M de Raadt, J Hamer, M Hamilton, R Lister, M Petre, D Tolhurst, & J Tutty (2006). The ability to articulate strategy as a predictor of programming skill. Eighth Australasian Computing Education Conference, ACE 2006, Hobart, Australia.

Simon, S Fincher, A Robins, B Baker, I Box, Q Cutts, M de Raadt, P Haden, J Hamer, M Hamilton, R Lister, M Petre, K Sutton, D Tolhurst, & J Tutty (2006). Predictors of success in a first programming course. Eighth Australasian Computing Education Conference, ACE 2006, Hobart, Australia.

Simon, J Sheard, A Carbone, M de Raadt, M Hamilton, & R Lister (2008). Classifying computing educaton papers: process and results. Fourth International Computing Education Research Workshop (ICER 2008), Sydney, Australia.

P Summons & Simon (1998). Authentication strategies for online assessment. Third Australasian Computer Science Education Conference, ACSE 1998, Brisbane, Australia.

D Tolhurst, B Baker, J Hamer, I Box, Q Cutts, M de Raadt, S Fincher, P Haden, M Hamilton, R Lister, M Petre, A Robins, Simon, K Sutton, & J Tutty (2006). Do map-drawing styles of novice programmers predict success in programming? A multi-national, multi-institutional study. Eighth Australasian Computing Education Conference, ACE 2006, Hobart, Australia.

D Valentine (2004). CS Educational Research: A Meta-Analysis of SIGCSE Technical Symposium Proceedings. Proc. 35th SIGCSE Technical Symposium on Computer Science Education, ACM SIGCSE Bulletin, 36, 1, 255-259.

J Whalley, C Prasad, & PKA Kumar (2007). Decoding doodles: novice programmers and their annotations. Ninth Australasian Computing Education Conference, ACE 2007, Ballarat, Australia.

K Woodford & P Bancroft (2005). Multiple choice questions not considered harmful. Seventh Australasian Computing Education Conference, ACE 2005, Newcastle, Australia.

J Zobel (2004). “Uni cheats racket”: a case study in plagiarism investigation. Sixth Australasian Computing Education Conference, ACE 2004, Dunedin, New Zealand.

Surely We Must Learn to Read before We Learn to Write!

Simon
University of Newcastle, Australia
simon@newcastle.edu.au

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

Ken Sutton
Southern Institute of Technology, New Zealand
ken.sutton@sit.ac.nz

Tony Clear
AUT University, New Zealand
tony.clear@aut.ac.nz

Abstract

While analysing students' marks in some comparable code-reading and code-writing questions on a beginners' programming exam, we observed that the weaker students appeared to be able to write code with markedly more success than they could read it. Examination of a second data set from a different institution failed to confirm the observation, and appropriate statistical analysis failed to find any evidence for the conclusion. We speculate on the reasons for the lack of a firm finding, and consider what we might need to do next in order to more thoroughly explore the possibility of a relationship between the code-reading and code-writing abilities of novice programming students.

Keywords: reading programs, writing programs, novice programmers.

1 BRACElet, reading, and writing

For many good writers
my heart has been bleeding
when beginners would try
to learn English by reading.

Forgive me, good readers
whom I may be slighting
in my selfish attempt
to learn English by writing.

(Piet Hein, Danish scientist, mathematician, and poet)

It is intuitively obvious that one cannot learn to write until one has learnt to read. However, many things that are intuitively obvious are in fact wrong. While we generally accept that this obvious rule applies to reading and writing natural languages such as English, we have chosen to explore whether it applies equally to reading and writing a programming language. We explore this question because, while it seems obvious to us that one cannot learn to write without first learning to read, many of our students appear not to agree.

BRACElet (Whalley et al., 2006, Clear, Edwards et al., 2008) is a multinational multi-institutional empirical study of students in introductory programming courses, whose overall purpose is to better understand how novices learn to read and write computer programs. A recent BRACElet initiative has been to ensure that programming exams include comparable reading and writing questions, with a view to analysing the nature of students' answers to these questions. In this paper, though, we concentrate on a purely quantitative analysis of the students' marks on these questions.

Expressed formally, the question that we hope to answer is this: does a novice programmer need to be able to read program code in order to be able to write program code?

2 The questions

The data analysed in this paper come from two different examinations at two different institutions: a final exam at a polytechnic in New Zealand and a final exam at a university in Australia. Nevertheless, both instruments included program reading questions and program writing questions that their creators felt were of similar difficulty, and both therefore provide useful data for this study.

If the courses were identical, we would have a single set of data and little confidence in its generality. But at the very least these two courses differ in the nature of the institutions at which they are taught, in the programming language used to teach them, and in the nature of their exams. Therefore any interesting results arising from them have a greater chance of being generalisable.

2.1 The university final exam

The university course, an introductory programming course for students in an IT degree, uses Visual Basic as its language of instruction. The selected questions from this exam can be summarised as follows.

Q23 (reading, 5 marks): explain the purpose of a loop, which counts the number of non-zero values in an array of integers.

Q25f (reading, 3 marks): explain the purpose of a loop that adds a predefined number of integer inputs and then divides by that number.

Q22 (writing, 5 marks): display the squares of successive integers, each on its own line, starting with the square of 1, until it has displayed all of the positive squares that are less than 1000.

Q24a (writing, 5 marks): append a family name, comma, space, and other names, unless both family name and other names are blank, in which case an error message is displayed.

Copyright © 2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

2.2 The polytechnic exam

The polytechnic course, also an introductory programming course for computing students, uses Java as its language of instruction. The selected questions from this exam can be summarised as follows.

Q4 (reading, 2 marks): generate the output produced by a loop involving an array and string concatenation.

Q5 (reading, 2 marks): generate the output produced by a loop involving an array and an if statement with &&.

Q6 (reading, 2 marks): generate the output produced by a loop involving an array and an if statement, and infer the purpose of the loop.

Q7 (reading, 3 marks): generate the output of part of a sorting program, infer the purpose of the part, and recognise the purpose of the whole.

Q8 (reading, 2 marks): generate the output produced by a loop that finds the maximum element in an array, and infer its purpose.

Q9 (reading, 2 marks): generate the output produced by a loop that finds the index of the minimum element in an array, and infer its purpose.

Q10 (reading, 2 marks): generate the output produced by a loop involving an array and a while statement with &&.

Q11 (reading, 2 marks): generate the output produced by a method involving a parameter, an array, and loop and break statements, and infer its purpose.

Q14 (reading, 1 mark): select from a list a code segment equivalent to the one provided, which includes an if statement and boolean variables.

Q16 (reading, 1 mark): deduce the value of a variable after completion of a loop that involves arrays and an if – then – else if – then – else statement.

Q12 (writing, 3 marks): write a nested loop involving print and println to produce a square of asterisks.

Q13 (writing, 4 marks – a Parsons problem): unjumble the provided lines of code to write a segment that finds the average of the numbers in an array.

2.3 Comparability of the questions and marks

Whether the reading and writing questions are really comparable is a vexed question. Of course there are elements of comparability: for example, on the university exam Q23 asks students to read and understand a simple loop, and Q22 asks them to write a simple loop. But there are also necessarily differences, as it would (probably!) not be reasonable to ask students in one question to read and understand a given piece of code, and in another to write the same piece of code. So while Q23 uses a for loop and includes an if statement, Q22 involves a while loop and the production of successive lines of output.

The lecturers of the two courses involved have determined that these questions are in some sense comparable – or, for this study, that the selected set of reading questions is reasonably comparable to the selected set of writing questions – but we have no objective scale of difficulty with which to measure each question and verify that determination.

In addition, reading questions and writing questions will often be marked differently. One of the early questions explored by BRACElet was whether successful reading of a code segment could entail tracing the code line by line, or whether it necessarily entailed understanding the purpose of the code (Lister et al 2006, Whalley et al 2006). With this distinction in mind, both of the university reading questions and about half of the polytechnic reading questions have marks allocated for explaining the purpose of the code, a criterion that really tends to be all or nothing.

On the other hand, it might be relatively easy to get part marks for a code-writing question. Traynor et al. (2006) quote a student as saying “Well, most of the questions are looking for the same thing, and you usually get the marks for making the answer look correct. Like if it’s a searching problem, you put down a loop, and you have an array and an if statement. That usually gets you the marks ... Not all of them, but definitely a pass.”

Denny et al. (2008) also comment on the nature of code-writing rubrics and their potential deficiencies. The analysis that follows must be considered in the light of these issues.

2.4 Reading as a precursor to writing: arguments for and against

We remarked in the introduction that while we believe a programming novice must learn to read before learning to write, many of our students do not agree.

Students in the university course addressed in this paper were given a weekly sheet of exercises. As has always been the case in this course, most of the exercises involved writing code. However, to help prepare the students for the reading questions on the exam, this year almost every sheet started with a single question that asked the students to explain the purpose of a code segment that was provided, or to answer other questions that would show their understanding of the code. Almost without fail, students would skip this question and go straight to the first code-writing question. When asked why, some of them explained that it was a lot easier to write code, with compilation error messages to help point the way, than to read it.

While no BRACElet publication has yet addressed the question as we are doing here, others have touched on it, without providing definitive answers. For instance Denny et al. (2008) found that a code writing task was easier than a code tracing task, but that there was a correlation between code tracing and code writing (Spearman’s r squared .37). Similarly, Lopez et al. (2008) found that the correlation between the ability to explain code and writing was positive (Pearson correlation .5586 at the 0.01 significance level). The path diagram proposed in Lopez et al. (2008) further suggested strong relationships between code ‘reading’ and ‘writing’ (as defined in the paper), but without necessarily implying a direction between the relationships. So the causation remains to be clarified.

3 The analysis

What we have done for this analysis is compare students’ marks on the reading questions with their marks on the supposedly comparable writing questions. Because there are different numbers of marks available for the two sets of questions, we first gave each student a single percentage mark for the reading questions and a single percentage mark for the writing questions. We then compared these marks for each student.

3.1 First findings

We began by looking at the correlation between the university students' reading and writing marks. We found a strong correlation ($r_{(27)}$=0.6543; p<0.01; adjusted R^2=41%) between these which is consistent with the studies mentioned above. Such a correlation, however, tells us nothing about the direction of possible causal relationships. To explore the causality, we carried out a visual inspection of the relationship between the marks.

Our first finding, shown in figure 1, stunned us. The figure shows a plot of the university students' writing marks against their reading marks. If students need to be able to read before they can write, we would expect the reading marks to be higher than the writing marks, and the bulk of the points to lie below the diagonal. This is indeed the case with the higher-scoring students, those to the right side of the plot; and this indeed makes sense for those students, many of whom scored full or nearly full marks for the reading, while falling a little short of that mark for the writing.

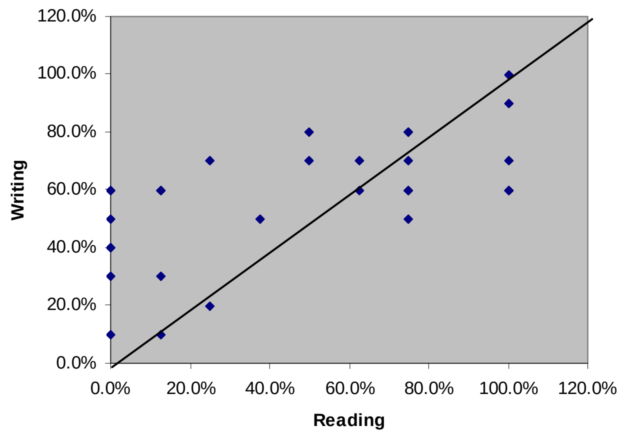


Figure 1: writing mark vs reading mark in the university course

The left side of the plot, though, tells a different story. The weaker students are all close to the diagonal or above it, some of them well above it, suggesting that they can write code at least as well as, and in many cases far better than, they can read it.

As a first attempt to confirm this trend, we took the difference between each student's reading and writing marks and plotted it against the student's overall course result (figure 2). There is a clear trend to this plot: the worst students (those on the left side of the plot) clearly perform better on the writing tasks than on the reading tasks. Dividing the class somewhat arbitrarily into terciles (or thirds), the bottom tercile (the worst third of the students) average 30% better on writing than on reading; the middle tercile average just 12% better on writing than on reading; and the top tercile average 7% worse on writing than on reading.

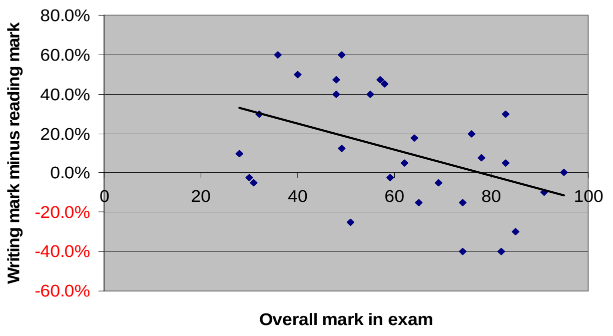


Figure 2: writing-reading difference against overall exam result, with trend line (university)

The data set is quite small, limited to those few students in the course who gave their consent to have their results analysed for this project; but an informal glance at the other hundred or so students in the course suggests that this pattern is reasonably representative of the whole class.

This was a somewhat startling discovery, running completely counter to our intuitive belief that students who perform poorly overall, and indeed students who perform poorly on code-reading tasks, will also perform poorly on code-writing tasks. We appear to have discovered that students can write before they learn to read.

3.2 A second data set

If the effect we observe here is real, we would expect it to be evident across disparate data sets. Having access to a second data set, that for the polytechnic course, we proceeded to examine that data in the same way.

In this dataset we also found a strong correlation ($r_{(22)}$=0.9056; p<.01; adjusted R^2 = 81%) between reading and writing marks, This correlation was surprisingly high, but it should be noted that the writing mark included a Parsons-type question (Parsons & Haden, 2006) which the literature suggests may be an intermediate level between reading and writing.

The initial plot of reading scores against writing scores (figure 3) is rather better clustered about the diagonal than the plot of figure 1, with no suggestion that the weaker students do better at writing than at reading.

The subsequent plot of the reading-writing difference (figure 4) actually shows a slight trend in the opposite direction to that shown in figure 2, suggesting that

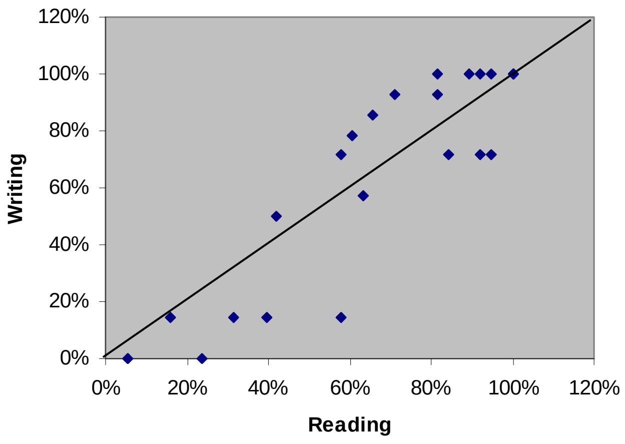


Figure 3: writing mark vs reading mark in the polytechnic course

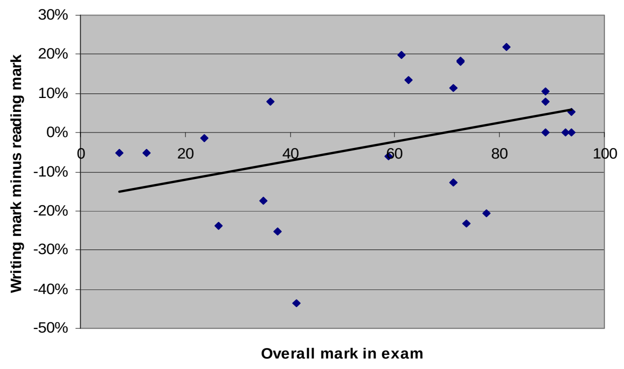


Figure 4: writing-reading difference against overall exam result, with trend line (polytechnic)

students do in fact need to learn to read before they can learn to write.

3.3 Statistical analysis and z-scores

The apparent finding from the university results is counterintuitive, but appears quite strong. However, similar examination of the polytechnic results fails to confirm the finding. As both of these findings arise from inspection, it is clear that some rather more rigorous analysis is required.

Our next step was to standardise the values that we are comparing.

On the university test, the reading questions are marked out of 8 and the writing questions out of 10. The difference between a student's writing and reading marks is therefore a higher proportion of the reading mark than of the writing mark. We try to compensate for this by considering the student's marks in each section as percentages of the possible mark, but this introduces a new problem: the item discrimination of questions may vary, resulting in a wider spread of marks for some questions than others. What we need is a way of using the same measuring scale for both sets of marks.

The solution to these problems is to standardise the data, which can be done in various ways, and which we have done in this instance by calculating z-scores for the marks. The z-score of a data point indicates how far it is from the mean of the data set, measured in units of standard deviations. A z-score of 1 indicates a point that is one standard deviation from the mean, and –1.5 indicates a point that is one and a half standard deviations away in the opposite direction. The z-score for a point *mark* is therefore simply *(mark – mean)/StDev*.

Figure 5 shows the same data as figure 2, but with the percentage differences replaced with the corresponding z-scores. The scatter of the points is now a fine example of random noise, and the trend line (which is included in the figure) is almost exactly horizontal and virtually indistinguishable from the z-score axis. Standardising the data has removed any suggestion of a meaningful relationship between writing-reading difference and exam result.

Figure 6 shows the same data as figure 4, but again with the percentage marks replaced by z-scores. Again the marks display a random scatter, and again the trendline is virtually indistinguishable from the horizontal axis.

Our first look at figure 1 suggested that poor students

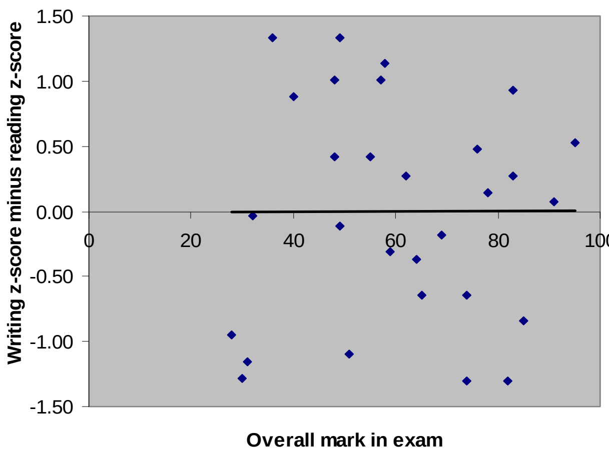


Figure 5: writing-reading z-score difference against overall exam result, with trend line (university)

do better at code writing than at code reading, while better students do better at code reading than at code writing. We appeared to have discovered some evidence that students do in fact learn to write before they learn to read. Our first look at a second data set failed to support the conclusion, and a proper statistical consideration of the results removed the suggestion entirely.

4 Discussion and threats to validity

Although in the end we have found nothing, we need to consider why this is so, and what we might need to do in the future if we are to discover any causal relationship between code-reading and code-writing skills.

Why is a causal relationship of interest? Because it can have a major impact on the way programming is taught to novices.

While we cannot find published references on the topic, we believe that the 'typical' programming course entails teaching people to write programs, in the assumption that learning to read programs will necessarily keep pace. But now and then an instructor receives a shock with the realisation that many students cannot even read programs, let alone write them (Lister et al., 2004). A likely consequence of this realisation is a shift in teaching approach to one that emphasises reading and understanding of code before going on to writing it. On the other hand, if we had indeed found that people can write programs well without being able to read them, there would be no need to devote any effort to teaching

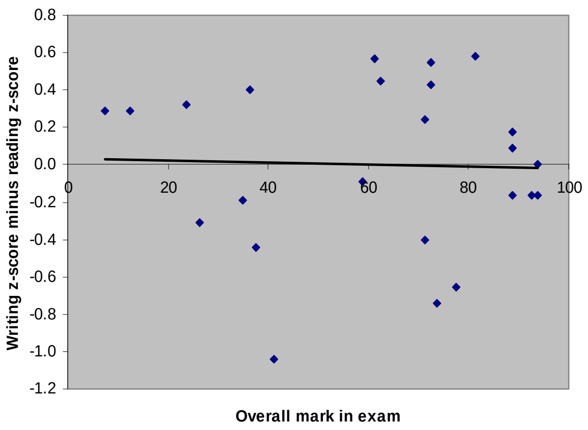


Figure 6: writing-reading z-score difference against overall exam result, with trend line (polytechnic)

the latter skill – and we would be arguing that it was inappropriate to examine program-reading skills in a course designed to teach program writing.

4.1 Comparability of reading and writing questions

Are the reading and writing questions in these exams really comparable? BRACElet has devoted some time to exploring the SOLO taxonomy (Lister et al., 2006, Clear, Whalley et al., 2008, Sheard et al., 2008) as one consideration in assessing the difficulty level of code-reading questions (Whalley et al., 2006), but we are not aware of any work that has been done to assign difficulties at the micro level. Is an assignment statement easier or harder to read and understand than a print statement? Is a nested loop easier or harder than an if-then-else? Is the difficulty of a piece of code simply the linear sum of the difficulties of its constituent parts? Does the difficulty depend on how familiar the student is with the construct? Without answers to these questions, the comparability of two code-reading questions is ultimately subjective and intuitive.

Further, while BRACElet has begun to explore the extension of SOLO to code-writing questions, or possibly the design of a different taxonomy for code-writing questions (Clear, Philpott et al., 2008), that exploration is far from complete, we have no reliable measure of the difficulty of code-writing questions even at the macro level, and so we are clearly not in a position to make any confident assertions about the comparability of particular code-reading and code-writing questions.

This problematic situation for assessing and understanding code writing in turn reflects earlier comments made concerning code reading.

> "We have a limited theory base relating to the comprehension of code … Our state of the art seems akin to studying reading without the key notions of comprehension level or reading age". (Clear, 2005)

Likewise, there appear to be no equivalent complexity metrics for code writing. A formal code complexity measure such as cyclomatic complexity (McCabe, 1976) may make a possible contribution, but such metrics tend to be designed to assess an already written large body of code, rather than the normally small fragments of an introductory programming course.

4.2 Comparability of the marking of reading and writing questions

If we do manage to overcome the difficulty of assessing the comparability of reading and writing questions, we will then face the related but distinct question of the comparability of their marking. It appears to us that the student quoted at the end of section 2.3 has a valid point: it might well be easier to get 'enough' marks on a writing question than on a reading question.

Following earlier work by the Leeds group (Lister et al., 2004) and BRACElet (Lister et al., 2006), we were conscious in our exam questions of the distinction between explaining a code segment line by line and explaining its overall purpose. Both are evidence of an ability to read the code, but the wording of many of our reading questions, and consequently our marking schemes for them, clearly emphasise the importance of the latter view, the big-picture understanding of the code.

It is not clear to us that this distinction between line-by-line understanding and big-picture understanding has a parallel in code-writing questions. If it had, it would presumably be along the lines that the more abstract an answer, so long as it was correct, the more marks it would earn. A correct answer in precise and accurate code would earn fewer marks than a higher-level pseudocode answer; and, taken to its logical extreme, the highest mark would go to the answer that merely rephrased the question, this being the answer that best expresses the overall purpose of the code in what SOLO calls a 'relational' manner.

Therefore the ideal that we have been setting for code-reading questions, that of big-picture understanding of the overall purpose of the code, is clearly one that we would be reluctant to apply to code-writing questions. This being the case, we are at present unclear as to how we might actually go about determining any comparable measures of student's code-reading skills and of their code-writing skills.

5 Future work

Where to from here?

We might be able to engage in a more detailed marking analysis of these data sets, one that would distinguish between the students' ability to trace code and their ability to perceive its overall purpose. This would be possible for some of the questions in the second data set, which asked for both tracing and overall-purpose answers, but not for those in the first data set, which asked only for the latter.

Particularly if that more detailed analysis is not possible or not fruitful, we could consider setting further exam questions that distinguish more clearly between the different reading-related skills, for example by asking students both to explain a piece of code line by line and to explain its overall purpose.

If we are to have a reliable measure of students' abilities at reading and writing code, we would need to consider a minute analysis of the difficulty levels of code-reading and code-writing questions at the micro level.

We would also need to continue our attempts to find a difficulty measure for code-writing questions comparable to the SOLO taxonomy for code-reading questions.

All of this should be done with an open mind and an acknowledgement that perhaps our counterintuitive non-finding is right. Many years ago, Sylvia Ashton-Warner (1963) proposed that teaching children to *write* English might be a better way of teaching them to *read* English than the traditional approach. Her method, which proved very successful with New Zealand Maori children, has not taken the world by storm, but that doesn't mean it is entirely without merit. Perhaps Piet Hein was not joking when he wrote the grook with which this paper began. Until we have clear evidence one way or the other, we should keep an open mind about whether our students need to be able to read code in order to be able to write it.

6 Acknowledgements

We are grateful to Raymond Lister for his founding and driving role in the BRACElet project, and for the specific ideas that he contributed to the discussion of this work; to Barbara Boucher Owens for reminding us of Sylvia Ashton-Warner's ideas about teaching children to read by teaching them to write; and to Mats Daniels for alerting us to the grooks of Piet Hein.

7 References

Ashton-Warner, S. (1963). *Teacher*. Simon & Schuster, New York.

Clear, T. (2005, Jun). Comprehending Large Code Bases - The Skills Required for Working in a "Brown Fields" Environment. *SIGCSE Bulletin* **37**:12-14.

Clear, T., Edwards, J., Lister, R., Simon, B., Thompson, E., & Whalley, J. (2008). The teaching of novice computer programmers: bringing the scholarly-research approach to Australia. *Tenth Australasian Computing Education Conference (ACE2008)*, Wollongong, Australia, 63-68.

Clear, T., Whalley, J., Lister, R., Carbone, A., Hu, M., Sheard, J., et al. (2008). Reliably Classifying Novice Programmer Exam Results using the SOLO Taxonomy. In S. Mann & M. Lopez (Eds.), *21st Annual NACCQ Conference* (Vol. 1, pp. 23-30). Auckland, New Zealand: NACCQ.

Clear, T., Philpott, A., Robbins, P., & Simon (2008). *Report on the Eighth BRACElet Workshop* (BRACElet Technical Report No. 01/08). Auckland: Auckland University of Technology.

Denny, P., Luxton-Reilly, A., & Simon, B. (2008). Evaluating a New Exam Question: Parson's Problems. In R. Lister, M. Caspersen & M. Clancy (Eds.), *The Fourth International Computing Education Research Workshop (ICER '08)*. Sydney, Australia: ACM.

Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004). A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *SIGCSE Bulletin* **36**(4):119-150.

Lister, R., Simon, B., Thompson, E., Whalley, J., & Prasad, C. (2006). Not seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In M. Goldweber & R. Davoli (Eds.), *The Tenth Innovation and Technology in Computer Science Education Conference (ITiCSE)* (pp. 118-122). University of Bologna, Bologna: ACM.

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. In R. Lister, M. Caspersen & M. Clancy (Eds.), *The Fourth International Computing Education Research Workshop (ICER '08)*. Sydney, Australia: ACM.

McCabe, T.J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering* **SE-2**(4):308-320.

Parsons, D. & 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.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., & Whalley, J. (2008). Going SOLO to Assess Novice Programmers. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '08)* (pp. 209-213). New York: ACM Press.

Traynor, D., Bergin, S., & Gibson, J.P. (2006). Automated Assessment in CS1. *Eighth Australasian Computing Education Conference (ACE2006)*, Hobart, Australia, 223-228.

Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P., et al. (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.

A People-First Approach to Programming

Donna Teague
Queensland University of Technology
Brisbane, Australia

d.teague@qut.edu.au

Abstract

Students continue to find learning to program difficult. Failure rates from introductory programming units are high, as are attrition rates from IT courses.

Case studies were conducted in 2007 involving Queensland University of Technology (QUT) introductory programming students who took part in weekly interviews and focus groups, and responded to questionnaires. Students divulged details relating to their attitude and approach to study, together with the level of confidence they had in their ability to learn to program.

Four of the case studies are included in this paper which portrays students with varying levels of confidence motivation, determination, attitude and study ethic, and how they each struggle to learn to program. The purpose of the studies was to determine to what extent each of these factors has an influence on student learning outcomes.

The studies focus on the people rather than the more traditionally studied cognitive difficulties of learning to program. The data collected from the case studies give some insight into the social barriers on many levels that students face and how they are dealt with and in some cases overcome.

The paper concludes with a discussion on student programmer personas as a design taxonomy and pedagogical tool.

Keywords: learning to program; student perceptions; motivation; determination; confidence; personas

1 Introduction

Queensland University of Technology is not alone in suffering a dramatic nose dive in enrolments in its Information Technology (IT) degree course over recent years. Attrition from similar courses worldwide is high (Seymour E et al. 1997; Kinnunen et al. 2006; Biggers et al. 2008), particularly for women and other minority groups for whom there is often poor representation to begin with (Cohoon 2002; Fisher et al. 2002; Lewis et al. 2006; Murphy et al. 2006; Reges 2006; Varma 2006; Vilner et al. 2006).

Commonly offered as a first year core subject, introductory programming subjects have an alarming failure rate (Sheard et al. 1998; Robins et al. 2003). Since 2003 an average of 31% of students were failing QUT's introductory programming subject (Teague et al. 2008).

Much work has been done in an attempt to address this learning to program dilemma. Cognitive theories include the difficulty of understanding the purpose of programs and their relationship with the computer; difficulty grasping the syntax and semantics of a particular programming language (Robins A et al. 2003); misconceptions of programming constructs (Soloway E et al. 1989); inability to problem-solve (McCracken M et al. 2001); and inability to read and understand program code (Lister R et al. 2004; Mannila L 2006).

But what of the students themselves? Who are they? Where do they come from? What is their attitude to study? What are their perceptions of learning to program?

This paper reports on four case studies involving introductory programming students and gives some insight into the diverse range of student attitude, motivation and self-confidence, all of which seem to be crucial elements for success. Three of the students were interviewed each week during the course of their study and the case studies document their attitudes and approaches to study and a perception of their weekly progress. The fourth case study results from a single interview with a student with a history of learning difficulties and failure. Some of these students provided further details towards their profile during a focus group session and/or by way of on-line questionnaire.

The objective of these and other case studies undertaken with introductory programming students was to get a better understanding of the social and cultural barriers that students face learning to program. After identifying and attempting to quantify a number of student characteristics including confidence level, motivation and perceptions regarding the learning material, the task was to determine to what extent these characteristics have an influence on student learning outcomes. In this paper, 'motivation' refers to the source of incentive to succeed and 'determination' refers to amount of commitment to that goal.

The results of this study may help educators identify and nurture the positive influences and breakdown the barriers of learning to program.

2 Related work

It has been said that confidence can play a significant role in the successful outcome of students learning to program (Gonzalez 2006). However, others report that lower confidence levels are not correlated to lower overall achievement (Murphy et al. 2006).

One of the major reasons for students to drop out of IT courses was found to be motivation (Kinnunen et al. 2006). Much effort has gone into developing courses and tools which aim to motivate and captivate introductory

Copyright (c)2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology, Vol. 95. Margaret Hamilton and Tony Clear, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

programming students and make learning fun (Lister 2004; Parsons et al. 2006; Pollard et al. 2006; Davis et al. 2007; Feinberg 2007)

Pair programming in the learning environment is one approach which addresses many of the issues that students struggle with while learning to program including not only those of a cognitive nature, but social and cultural barriers as well. Students find programming in pairs creates a social rather than competitive environment which promotes interaction and lends twice the cognitive resources and an extra set of eyes to a programming exercise (Simon et al. 2007). Studies on collaborative learning and pair programming identify some of the issues that students face, and this paper documents how those issues may manifest while learning to program.

3 Case Studies

In order to understand what barriers may exist for students learning to program and why they often fail, amongst other things we need to investigate how students approach learning to program and what their attitudes to study are. Case studies are described as the preferred strategy for considering these types of questions (Yin 2003).

Volunteers were sought from those students enrolled in either of the first year programming units at QUT in semester 2 of 2007 who were willing to discuss their ongoing experiences of learning to program. Weekly interviews ensued for the duration of the 13 week semester with eight such students. Although pursuing a consistent line of enquiry each week, interviewees were given ample time to discuss whatever seemed important to them, as well as having the opportunity to take advantage of the interviewer being a member of the teaching staff from whom technical assistance could be sought. After an initial ‘getting to know you’ and trust-building period, interviews soon became relaxed non-threatening sessions where students seemed comfortable discussing their experiences in an open and honest manner, without fear of judgement or retribution.

The data collected from these guided conversations, complemented with survey responses and focus group input, form the basis of what Bassey (1999) refers to as ‘theory seeking’ case studies. Four of those case studies are summarised in this paper.

The two units involved:

ITB001 *Problem-Solving and Programming* is QUT’s first programming unit, a core unit where students are introduced to solving computational problems and implementing solutions (at the time of this experiment in Scheme). This unit has no prerequisites, but typically the student cohort tends to have a wide variation of computer and programming skills ranging from very limited experience with a computer, to a small amount of programming industry experience. Most ITB001 students are doing their first semester of the Information Technology degree, although there is often a proportion of second-year double-degree students.

The assessment for this unit consisted of three individual assignments of increasing difficulty (total of 50%) and an end of semester written exam (50%).

ITB003 *Object Oriented Programming* is the second programming unit for QUT, with ITB001 as its prerequisite. This unit builds on the skills developed in ITB001, focusing on understanding and implementing an object-oriented design specification in C#. ITB003 is a second semester, first year unit with a high proportion of students who have chosen a software engineering major.

The assessment for this unit consisted of weekly on-line quizzes (10%), a semester-long individual programming project submitted in two phases (30%); short answer exam-like review questions (10%), and an end of semester written exam (50%).

Each unit represented 25% of a full-time study workload, and both conducted a one to two hour lecture and a two hour workshop each week. Workshops involved students completing programming exercises to reinforce in a practical way the material previously introduced in a lecture. Attendance at workshops was strongly encouraged but was neither obligatory nor counted towards final grades. Apart from lectures and workshops, all students were expected to dedicate an extra 8 or 9 hours per week to self-directed study for a total of 12 hours study per unit per week. The students involved in the case studies were asked how much time they *actually* spent on the unit.

Students were asked similar questions each week including how difficult or otherwise they were finding the unit, and their current enjoyment and confidence levels. Students graded each of these perceptions on a Likert scale of 1 (low) to 5 (high).

The final grade awarded to each student is included in this paper, on a scale of 1 (low) to 7 (high) with 4 being a passing grade.

3.1 Nelly [studying ITB001]

3.1.1 Profile:

Nelly is a domestic female student enrolled full-time in the Bachelor of IT course, majoring in Software Architecture. She is 27, lives in shared accommodation with others of similar age and averages about 21 hours a week paid work. Nelly was interviewed during her first semester of university while studying her very first programming unit, and had no previous programming experience except “dabbling in HTML”.

Nelly has a history of personal challenges, but in recent years had the determination to fight and overcome a family predisposition for a debilitating health condition.

3.1.2 Pre-ITB001

Perception of:	***Scale of 1 – 5*** *(1 = low; 5 = high)*
Confidence	4
Enjoyment	4
Difficulty	3

Table 1 – Nelly: Pre-Unit Perceptions

Nelly was looking forward to learning to program, but felt at a disadvantage because she didn’t fit the stereotype: no programming experience, and weak math skills.

She sees programming as something for ‘nerds’, but also adds that ‘nerds are cool’ and for that reason, doesn’t mind being classified as one if she does well in this unit. Nelly strongly agreed that it makes sense that there are more men than women in programming.

3.1.3 During ITB001

Average hours spent on ITB001 per week	12
Average hours spent on un/paid work per week	21

Table 2 - Nelly: Study versus Work

Nelly advised that she suffered from anxiety and lack of sleep, and identified this as a distraction to her studies. In weeks where she didn't dedicate the required 12 hours of study to the unit, it was as a result of lack of time to do so.

Nelly's confidence during the course of the semester was volatile. She generally welcomed a challenge, and when presented with a difficult topic her approach was to simply try harder, put in more time and keep practicing until it 'sunk in'.

> *...Need more practice mastering this topic. Need more exercises. But it all makes sense. … It gets so confusing. …Recursion is doing my head in and you get frustrated with it. I think I have it - then do the next exercise and I haven't!*

Nelly persisted with hands-on practice even though some topics were 'frustratingly challenging'. She often reported serious self-doubt in these situations, but was convinced that persistence would eventually pay off.

> *[I was] stuck on problems for so long. I'm stupid - I'm not getting it. Takes me so long to figure things out. I eventually get it though. I'm a kinesthetic learner. The [iteration] exercise really got me. I was so pleased when I figured it out.*

Nelly rarely sought help from the teaching staff when she was stuck, because she felt like they were "too important to bother", and felt it would be an 'intimidating' experience. She also confessed to keeping up the pretence of being confident and capable, and feared that asking for help would destroy that image and make her appear weak and incapable.

She also expressed disappointment in her friends and peers that they didn't seem to be putting in enough effort – that they would attend the lectures and workshops, but do little further work themselves. Although preferring to work through problems by herself first, Nelly valued the role of collaboration in learning and ended up mentoring a couple of friends who were struggling with the unit, assuming a leadership role normally undertaken by paid peer mentors.

She always seemed to be up to date with the work and had completed most of the workshop exercises before the workshop, but attended them anyway out of fear of "missing something". This fear probably resulted from a need to prove her ability (to either herself or someone else) while harbouring some doubt that she could succeed.

Even when Nelly had successfully solved a programming problem, she was not confident in the quality of her solution. She admitted to being a bit of a perfectionist and showed a keen interest in seeing alternative approaches to the same problem.

> *It's a case of my usual problem solving strategy of ramming myself into the brick wall of a problem until enough pieces fall off to let it get through. … But it does the job, so I'm sort of happy. … It passes all the tests.*

3.1.4 Post ITB001

Perception of:	***Scale of 1 – 5*** ***(1 = low; 5 = high)***
Confidence	4 (stable)
Enjoyment	5 (up from 4)
Difficulty	4 (up from 3)

Table 3 - Nelly: Post-Unit Perceptions

Nelly enjoyed the unit more than she had predicted she would, even though she found it more difficult than expected. She summarised her motivation and attitude towards studying programming as wanting to do her "absolute best", and put in a big effort to do so.

After completing the exam, but prior to release of grades, Nelly reflected that overall she had been fairly confident of being about to successfully learn how to program during the semester, and predicted a final grade of 7. She said she loved programming and could not wait to do the next programming unit.

	Final grade
Expectation:	
- pre-ITB001	7
- pre-exam	7
Actual	7

Table 4 - Nelly: Grade Expectation versus Actual

3.2 Jane [studying ITB001]

3.2.1 Profile

Jane is a domestic female student enrolled full-time in the Bachelor of IT course, with (as yet) unchosen major. She is 20+ and works part-time, averaging about 5.5 hours a week. Jane failed her first attempt at this unit in the previous semester, which constituted her first taste of programming. She cited "lack of time" as the main factor for failing, and has since reduced her working hours as a result.

3.2.2 Pre-ITB001

Perception of:	***Scale of 1 – 5*** ***(1 = low; 5 = high)***
Confidence	2.5
Enjoyment	4
Difficulty	4

Table 5 - Jane: Pre-Unit Perceptions

Jane predicted that this would be the hardest unit for her this semester, but was confident she now had the time necessary to give it a good shot.

Jane has a very nervous disposition, and was constantly fidgeting. She lacked confidence in terms of both her university studies and personal life. A health condition was a constant distraction, which was exacerbated by elevated anxiety levels. Jane advised that she was constantly stressed, sometimes to the extent of having difficulty carrying out everyday tasks.

Jane confessed that her severe lack of confidence had in the past caused her to withhold assignments from submission because she believed them to be too inferior. She was also reluctant to consult with teaching staff about her assignments for the same reason.

Given her previous experience with learning to program, and her anxiety disorder, Jane had little confidence from the beginning that she could successfully learn to program.

3.2.3 During ITB001

Average hours spent on ITB001 per week	11
Average hours spent on un/paid work per week	5.5

Table 6 - Jane: Study versus Work

In the early weeks of the semester, although showing signs of a hectic social life and still coming to grips with her study timetable, Jane was finding the easier concepts a bit of a confidence boost:

I didn't really learn much last semester. I really like it now that I'm actually doing it. It's not difficult at all - and that's a huge relief. I was so worried!

Jane explained how anxiety affects her learning during workshops:

It would probably help if I had a look at [the exercise] in my own time. Because of my anxiety I can't actually understand - think straight at the time. It's really distressing for me. All I do is smile and nod at the time. I can't actually take it in.

There was little clue of Jane's panic and lack of progress during workshops as outwardly she presented as a vivacious student who rarely asked questions, and when approached, consistently confirmed that she was up to speed and happy with her progress.

Jane had a habit of accepting misinterpreted explanations of concepts without question, and without testing it or proving it to herself. This became evident when she spoke about her understanding of how a *non-recursive* function call works:

Like the computer goes through [the function] and unless you write code to stop it, it will just do it again. I didn't realise …that's how Scheme works. [Give me an example, tell me what you mean.] *Like once it's gone to the bottom of your code it goes back up and starts again at the top.* [But only if there is a recursive call]. *Oh. Really?* [So if your procedure says add 2 plus 3, it will add 2 plus 3 then stop] *Really? Well that explains THAT then! What makes it go around and do it again then?* [Recalling the procedure <demo ensued>..]

As the semester progressed, Jane convinced herself that her struggle was not with implementing code, but with the preparatory problem solving:

I understand writing procedures more than having to work out the problem. So if the problem was solved logistically, I could probably write the program.

She continued to have little faith in her ability.

I'd probably make mistakes even though I thought I got it right.

She reportedly spent a "ridiculous amount of time" on simple assignment tasks, sometimes being in the computer lab over night. She approached assignments with extreme caution, once she had actually worked up the courage to start. She was actually "scared" of doing them.

After consultation with counselling services, and on advice from the teaching staff, Jane delayed work on assignment tasks until she had thoroughly revised the workshop material which was designed to develop the skills required for the assignment[2]. She was then advised to spend a small amount of time on the assignment task, then seek help if she couldn't progress further.

Jane was delighted that this approach seemed to work for her by building her confidence:

I didn't even look at the assignment for 3 hours, I just went through all the other little exercises. Then once I got to the actual assignment, it only took me half an hour. So I thought - maybe I CAN do it. :-)

Jane continued to struggle through the semester, eventually dropping another unit to ease the workload. She attended two workshops for ITB001 each week, with the intention to use the first as a preview, then taking time to work through the material herself before the second workshop where she would be more confident with the material and less influenced by stress.

3.2.4 Post ITB001

Perception of:	***Scale of 1 – 5*** ***(1 = low; 5 = high)***
Confidence	3.5 (up from 2.5)
Enjoyment	5 (up from 4)
Difficulty	3.5 (down from 4)

Table 7 - Jane: Post-Unit Perceptions

Jane said that she had 'enjoyed this unit immensely' which seemed incongruous with the degree to which she had suffered with anxiety and lack of confidence and struggled with the learning as a result. Nevertheless, Jane's expectation of grade increased from 4 to 5 just prior to sitting the final exam.

	Final grade
Expectation:	
- pre-ITB001	4
- pre-exam	5
Actual	4

Table 8 - Jane: Grade Expectation versus Actual

3.3 Dave [studying ITB003]

3.3.1 Profile

Dave is a full-time domestic student and fits the stereotype of a programming student: recent school leaver, highly confident, limited social skills and a keen interest in technology. Unlike the previous two students, Dave was profiled during his second semester of university, while completing his second programming unit. Other previous experience included programming at school, being part of an internet website development

[2] This approach is continually recommended to students, but rarely adopted.

group; building circuitry, as well as a list of half a dozen programming languages that he was familiar with.

Dave lives at home with his family and is not employed in paid or unpaid work.

3.3.2 Pre-ITB003

Perception of:	*Scale of 1 – 5* *(1 = low; 5 = high)*
Confidence	4
Enjoyment	5
Difficulty	2.5

Table 9 - Dave: Pre-Unit Perceptions

Dave expected to enjoy this programming unit, but went to great lengths to convince me that he probably knew it all already. For that reason, he expected it to be fairly easy and was confident of a high overall grade.

Dave advised that he had an extraordinary memory and a mind for computing that meant he didn't need to write much down, nor work through any problem-solving or design process in order to implement an exemplary solution.

I can do a lot of the testing in my mind because I have a mind that can just keep track of variables - millions of variables and just watch code execute in my mind.

Can read and understand code in any language - even if I've never used that language before.

3.3.3 During ITB003

Average hours spent on ITB003 per week	5
Average hours spent on un/paid work per week	-

Table 10 - Dave: Study versus Work

Dave attended lectures and workshops with a friend he had gone to school with, but gave the impression that he need not have bothered, given his already significant programming ability. Dave measured his own confidence at the highest level (5) for most of the semester.

[I'm] not bothered to spend more time. Its just basic at the moment - will work more when it is harder. It just comes natural to me.

Dave's attitude to the semester-long programming project for assessment was that he would be able to complete it in a couple of weeks and therefore didn't need to spend the recommended time each of the 13 weeks of semester working on it. He was pretty sure he would be able to "knock most of it over" during the week-long semester break.

The first phase of Dave's project was graded 30/50 and he was happy with that result. Although his submission was incomplete:

[I] didn't do the test cases - couldn't be bothered. I'm lazy - I'll do the bare minimum…

He believed the true fault lay with the project specifications supplied to students which were "horrible…badly worded".

As to the unit as a whole, although his attendance at lectures and workshops was good, Dave seemed to have his own timetable and agenda:

Workshops (all units) go too slow. So I leave it to the end and teach myself.

During workshops, Dave would often take part in the activities, or contribute for a short while until he lost interest and until he found something more appealing to do. As a result, some tasks remained unsolved and it was not evident (except from his own insistence) that he was actually capable of completing them successfully.

3.3.4 Post ITB003

Perception of:	*Scale of 1 – 5* *(1 = low; 5 = high)*
Confidence	3.5 (down from 4)
Enjoyment	5 (stable)
Difficulty	3.5 (up from 2.5)

Table 11 - Dave: Post-Unit Perceptions

Dave reflected on the unit as follows:

Lectures would be useful if I listened - ok for basics.

He insisted that he required very little time to complete programming tasks, including the non-trivial programming project required for this unit. Dave seemed loath to spend time early in the semester on project tasks that he was confident he could complete in a short amount of time just prior to submission date. Combined with the fact that other units had similar assessment demands late in semester, this attitude resulted in insufficient time to complete all aspects of the project successfully. Nevertheless, Dave's confidence did not waiver and he continued to expect a high grade for this unit.

	Final grade
Expectation:	
- pre-ITB003	6
- pre-exam	7
Actual	4

Table 12 - Dave: Grade Expectation versus Actual

3.4 Steve [studying ITB001]

3.4.1 Profile

Steve is a 24 year old domestic student who has recently failed to complete the unit after a number of previous unsuccessful attempts and withdrawn enrolment. Steve presented as fairly confident about his academic ability, but socially inept.

Steve was frequently witnessed in both lectures and workshops listening to music, playing games and refusing to take part in either class discussions or to converse with teaching staff on a one-on-one basis. Given this attitude, his presence in class confounded both his teachers and fellow students. Steve agreed to be interviewed after completion of the unit, in order to help us understand his attitude towards study and his motivation for studying programming.

Steve's family has been in Australia for 17 years, and he is currently in the process of moving out of home.

Steve discussed a less than idyllic schooling where he was very shy, had trouble making friends and received a very poor senior high school result. He feels pressure from his mother to study, get a job and financially support the family. Since school, he has completed a Diploma at TAFE in order to gain entry to university. Steve expressed a keen interest in working in the computer gaming industry, but was disappointed that he hadn't as yet worked in IT at all. In fact, Steve had never had any kind of job, paid or unpaid.

Steve's academic history at university reflects a very poor result, with a number of units having been repeated, but none, as yet, completed successfully.

I think it's just me being lazy. That's all. It's doable.

Asked how best he learns, Steve responded that his preference was for active hands-on repetitive learning:

Yeah, doing. Information sort of sneaks out when I'm reading…If I'm doing it – it will still slip out but I can always repeat it because I'll know what to do. So I can just keep on repeating that action until I get it.

3.4.2 ITB001 studies

As Steve was not interviewed prior to completion of the unit like the previous three case studies, there is no record of his perceived confidence, enjoyment and level of difficulty of the unit.

Average hours spent on ITB001 per week	<1
Average hours spent on un/paid work per week	0

Table 13 - Steve: Study versus Work

Steve understood the workload requirements to successfully complete this unit, but dedicated very little time to study, if any. He admitted that if he was really serious about it, he would have to spend at least three hours per day on a unit.

On repeating a unit, Steve would convince himself he was way ahead, and lose the motivation to work:

I felt pretty confident – because actually I was getting ahead of the program. I actually stopped myself. I actually felt lazy. It felt good I was getting ahead but like it was a good excuse to let me do something else.

As to programming, Steve said he would have enjoyed it more if he had known what he was doing:

If I knew what I was doing, I would like it. But I don't. And I think if I pay more attention and focus on the work and not get sidetracked a lot, I would like it. I like working things out.

Steve blamed his lack of ability to learn programming on his attitude:

Like it's because I'm not paying attention at all. You need to be focused on it fulltime or else you will fall behind….I never really got into the knowledge of it. All the ins and outs of programming. Need to find out about it.

Further discussions about Steve's study ethic revealed that he was seriously distracted with games, and tended to play World of Warcraft at every possible moment, equating to around 16 hours each and every day:

[I play it] whenever I'm awake. It's really addictive.

Steve described the World of Warcraft environment as a social place where he interacted with his friends. Because he doesn't have any real people to "hang out" with, he logs on and hangs out there.

So if I can just get more real life friends that I can meet face to face I wouldn't have the need to go on World of Warcraft much.

Asked about his preference for learning environment, Steve said he preferred studying with a friend, or in a small group:

I want to interact because I want to get better at socialising and it gets lonely if I'm always by myself. You need to be talking with people anyway. It's good for your health. Right now I'm trying to do as much of that as possible.

Steve habitually logged on to the game as soon as he woke up, so it was no wonder his attendance at university and studies suffered. Although he showed a little reluctance to continue this habit, he described a sense of commitment to his virtual gaming friends:

Like, World of Warcraft – there's a community. So it's – I feel like I have an obligation to people in the game.

After moving house and no longer having internet access:

I don't [miss playing it]. …Relieved actually. Because I felt like I had to log on every time but now I don't have to, because I can't. So it's a relief that I don't have that obligation.

Steve admitted that his addiction to World of Warcraft had stopped him from being serious with his study. He also claimed he had kept up to date with this unit for the first three weeks of semester at which point he needed to take time off due to sickness, and then fell behind with his studies. He eventually withdrew from the unit.

3.4.3 Update

Steve re-enrolled in ITB001 in semester 1, 2008 (the following semester) and showed a little improvement in at least his attendance at and involvement in workshops. However, his focus was clearly set on completing the assessment items, and completely ignored the workshop exercises that were designed to introduce problem-solving and programming concepts and build the expertise required for the assignments. His demeanour continued to be that of an introvert; however it was evident that he was making a small effort to communicate with other students outside class. Steve started asking questions during workshops, but seemed to make little progress in terms of problem-solving skills and ability to program.

Steve regularly attended lectures and workshops as well as extra catch-up sessions that were made available for students falling behind. His assignment submissions indicated some small amount of progress in his understanding of the unit content, although a great deal of help had apparently been sought from many of the teaching staff, and we suspect also from other students. Steve sat the final exam for ITB001, but failed the unit.

Steve again enrolled in ITB001 in semester 2, 2008 and his attendance at workshops was initially very good. He was less distracted in class, and made a habit of asking many questions, not in group discussions, but privately with a tutor. After spending much time with Steve, it became apparent that one-on-one tuition was not only what he expected, but what he benefited from the most. After gaining a little confidence with his tutor, he verbalised his frustration with the wording of one particular task's instructions. After this was rectified, for the very first time Steve was able to complete an exercise by himself. He was praised for his efforts and it was quite obvious he was very proud of himself and continued with renewed confidence. Surprisingly, the following week he was called on to present his solution to the class, and did so without hesitation – another first. His solution was neat, well described and of fairly good quality.

His acute shyness and lack of social confidence had initially made it difficult for him to verbalise the difficulties he was having. As a result, he had chosen to struggle on by himself. What he is capable of achieving with a little confidence and encouragement remained to be seen.

Unfortunately, to the best of the author's knowledge, Steve attended no further workshops after the week he presented his solution to class. Perhaps the prospect of being called on again to address the class was too much to bear. Alternatively, the small progress he made may have pushed his confidence into overdrive, lending him to mistakenly believe he could complete the remainder of the unit without attendance. Steve, once again, failed the unit.

4 Discussion

Although in the interviews Nelly graded her confidence as fairly high, her confidence was clearly reliant on her ability to solve the current task at hand. She tended to over-prepare for workshops, by which time she had ensured that she had already acquired the necessary skills.

The transcripts of her interviews (some quotes from which are included in the previous section) gave a better insight into her personal struggle to succeed. Nelly's health struggle against difficult odds, showed the power of determination she had at her disposal. Faced with frustratingly challenging programming concepts, this determination was called into play. Again, she proved to herself and others that she was capable. Although not always confident, she valued the appearance of being confident and capable, which at times must have made it a difficult and lonely task for her to resolve issues with the unit. Determination was a key factor in Nelly's success.

Jane also showed a fierce determination throughout the unit, when it seemed like the odds were against her. She struggled with the unit content and anxiety issues, but never with motivation and attitude. Although close to what seemed like a nervous breakdown, she still happily declared that she loved programming and was determined to keep going.

Dave's profile highlighted all too often seen traits of young male students – competitive and over-confident yet lazy and unmotivated. Their definition of 'success' is probably just passing the unit, rather than excelling – or simply confirming their aptitude to themselves rather than proving it via formal assessment.

Steve is a bewildering case, and one which, if possible, will be followed up with some interest. It will be intriguing to see if a little more confidence, combined with a better work ethic could see him pass any units in the future.

The common issue with each of the four students is their level of confidence, motivation and determination. Their aptitude (which has not been documented) may actually be less relevant in terms of being able to successfully learn to program. One tool which addresses confidence and motivational issues is collaborative learning (Wilson et al. 1993; McKinney et al. 2006) and more particularly, pair programming where students enjoy significant educational benefits including active learning and improved retention, program quality, and confidence in the solution (McDowell et al. 2002; Williams et al. 2002; Nagappan et al. 2003; Werner et al. 2004; McDowell et al. 2006; Mendes et al. 2006).

There is a presumption by educators that university students aim to succeed: but 'success' is surely subjective. Students oozing confidence in their programming ability with a similarly matched weighty ego are not inevitably those who achieve high grades. Nor are they necessarily even motivated to do so. Similarly, students struggling with low confidence are not always low achievers. Stress and anxiety issues, often reported in particular by female students, if coupled with the motivation and determination to succeed, are not always insurmountable obstacles.

Some students can be reluctant to admit they are having difficulty and fear losing face. Others want to play the underdog and pretend they are having similar troubles as their mates, because to them it might be uncool to be smart.

Great aptitude and confidence do not, in isolation, guarantee superior marks. Without motivation there may be little reason to 'succeed', whatever definition you hold. If there is a lack of motivation to do well, a student's true ability may not necessarily be reflected in their overall result. Conversely, a lack of natural ability does not always result in poor results if the student has a good attitude and is determined enough.

The following table roughly plots the confidence and determination levels of those four students including their final results. The lighter shaded area of the table represents a feasible estimation of the required level of each attribute for successful outcome.

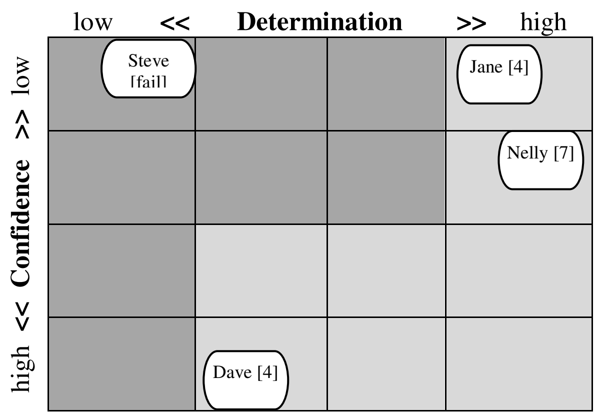


Table 14 – Confidence versus Determination

5 Conclusions

What this paper has highlighted is that *what you see* isn't always *what you get*.

Student final grades seem to be neither good indicators of ability, motivation, determination nor confidence. Equally, learning outcomes are not easily predicted based on confidence or ability.

The four students in these case studies reflect a huge variation in attitudes and provide a rich source of data relating to student attitudes and their perception of their level of confidence, enjoyment and difficulty of programming.

Confidence may be a key issue in successful learning outcomes – but is not in itself a reliable predictor of success. It seems that *determination* (ie the level of commitment to reach whatever goal the student is motivated to achieve) also plays a significant role in success, at any level of aptitude.

If the key to success in learning to program is a balance between aptitude, confidence and determination, ethnographical-like studies may be useful to develop a set of Personas, or fictitious characters, representing the likely range of student programmers in terms of these attributes. Personas are used in human computer interaction design to model user demographic and behavior and articulating user population. They represent *hypothetical archetypes* of actual users (Cooper 1999).

Importantly, it seems obvious from these studies that assumptions must not be made about students' likelihood of success when based on superficial assessments. Rather, we need to delve a little deeper than normal into the person behind the student, in order to determine the barriers most likely to affect their ability to learn to program.

Computer science educators could better understand the real cultural and social issues that characterise the *real people* learning to program, by developing precise descriptions of students, their perceptions and attitudes.

Programming student personas could combine the attributes of confidence and determination with learning styles, to become a design taxonomy for courses and a pedagogical tool for teaching and support. Personas could be plotted in a similar manner to Table 14 with each of the attribute dimensions quantifiable in some way. This pertinent information may enable timely intervention by teaching staff with students fitting the persona of a student at risk of failure due to ill-placed confidence, or lack of motivation and determination.

6 Acknowledgements

Thanks to the students who gave up their time to share cake and coffee with me (especially Nelly, Jane, Dave and Steve – *not their real names*), while reflecting on their program learning experiences at QUT.

7 References

Bassey, M. 1999. Case Study Research in Educational Settings. Buckingham, Open University Press.

Biggers, M., Brauer, A. and Yilmaz, T. 2008. Student Perceptions of Computer Science: A Retention Study Comparing Graduating Seniors vs. CS Leavers. 39th SIGCSE technical symposium on Computer science education, Portland, OR, USA, ACM.

Cohoon, J. M. 2002. "Women in CS and Biology." ACM SIGCSE Bulletin, Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education SIGCSE '02 34(1).

Cooper, A. 1999. The Inmates are Running the Asylum. Indianapoloois, IN, Sams.

Davis, J. and Rebelsky, S. A. 2007. Food-First Computer Science: Starting the First Course Right with PB&J. 38th SIGCSE Technical Symposium on Computer Science Education. Kentucky, USA.

Feinberg, D. 2007. A Visual Object-Oriented Programming Environment. 38th SIGCSE Technical Symposium on Computer Science Education. Kentucky, USA.

Fisher, A. and Margolis, J. 2002. "Unlocking the clubhouse: the Carnegie Mellon experience " ACM SIGCSE Bulletin 34(2).

Gonzalez, G. 2006. A Systematic Approach to Active and Cooperative Learning in CS1 and its effects on CS2. SIGCSE 2006 Technical Symposium on Computer Science Education. Houston, Texas, USA.

Kinnunen, P. and Malmi, L. 2006. Why Students Drop Out CS1 Course? 2006 international workshop on Computing education research ICER '06.

Lewis, S., McKay, J. and Lang, C. 2006. The Next Wave of Gender Projects in IT Curriculum and Teaching at Universities. Eighth Australasian Computer Education Conference (ACE2006), Hobart, Tasmania, Australia, ACS.

Lister, R. 2004. Teaching Java First: Experiments with Pigs-Early Pedagogy. 6th Australasian Computer Education Conference (ACE2004). Dunedin, Australian Computer Society Inc.

Lister R, Adams E, Fitzgerald S, Fone W, Hamer J, Lindholm M, McCartney R, Moström J, Sanders K, Seppällä O, B, S. and Thomas L 2004. "A Multi-National Study of Reading and Tracing Skills in Novice Programmers." SIGSCE Bulletin 36(4): 119-150.

Mannila L 2006. Progress Reports and Novices' Understanding of Program Code. 6th Baltic Sea Conference on Computing Education Research, Koli Calling.

McCracken M, Almstrum V, Diaz D, Guzdial M, Hagan D, Kolikant Y, Laxer C, Thomas L, Utting I and Wilusz T 2001. "ITiCSE 2001 working group reports: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students." ACM SIGCSE Bulletin 33(4).

McDowell, C., Werner, L., Bullock, H. and Fernald, J. 2002. The Effects of Pair-Programming on Performance in an Introductory Programming Course. 33rd SIGCSE technical symposium on Computer science education. Cincinnati, Kentucky ACM.

McDowell, C., Werner, L., Bullock, H. E. and Fernald, J. 2006. "Pair programming improves student retention, confidence, and program quality " Communications of the ACM 49(8).

McKinney, D. and Denton, L. F. 2006. Developing Collaborative Skills Early in the CS Curriculum in a Laboratory Environment. SIGCSE 2006 Technical Symposium on Computer Science Education. Houston, Texas, USA.

Mendes, E., Al-Fakhri, L. and Luxton-Reilly, A. 2006. A Replicated Experiment of Pair-Programming in a 2nd-year Software Development and Design Computer Science Course. ITiCSE 06: Proceedings of the 11th annual conference on Innovation and technology in computer science education Bologna, Italy.

Murphy, L., McCauley, R. and Westbrook, S. 2006. Women Catch Up: Gender Differences in Learning Programming Concepts. SIGCSE 2006 Technical Symposium on Computer Science Education. Houston, Texas USA.

Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C. and Balik, S. 2003. Improving the CS1 Experience with Pair Programming. 34th SIGCSE technical symposium on Computer science

Parsons, D. and Haden, P. 2006. "Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses." Eighth Australasian Computer Education Conference (ACE2006) 52: 157-163.

Pollard, S. L. and Duvall, R. C. 2006. Everything I Needed to Know About Teaching I Learned in Kindergarten: Bringing Elementary Education Techniques to Undergraduate Computer Science Classes. SIGCSE 2006 Technical Symposium on Computer Science Education. Houston, Texas, USA.

Reges, S. 2006. Base to basics in CS1 and CS2. SIGCSE’06, Houston, Texas, USA, ACM.

Robins A, Rountree J and Rountree N 2003. "Learning and Teaching Programming: A Review and Discussion." Journal of Computer Science Education 13(2): 137-172.

Robins, A., Rountree, J. and Rountree, N. 2003. "Learning and Teaching Programming: A Review and Discussion." Journal of Computer Science Education 13(2): 137-172.

Seymour E and Hewitt NM 1997. Talking About Leaving, Westview Press, Harper Collins Publishers.

Sheard, J. and Hagan, D. 1998. Our failing students: a study of a repeat group. Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education: Changing the delivery of computer science education ITiCSE '98.

Simon, B. and Hanks, B. 2007. First Year Students' Impressions of Pair Programming in CS1. Third International Computing Education Research Workshop. Georgia Institute of Technology, Atlanta, GA USA, ACM.

Soloway E and Spohrer J 1989. Studying the Novice Programmer. Hillsdale, NJ, , Lawrence Erlbaum Associates.

Teague, D. and Roe, P. 2008. Collaborative learning: towards a solution for novice programmers. Proceedings of the tenth conference on Australasian computing education. Wollongong, NSW, Australia, ACS.

Varma, R. 2006. "Making Computer Science Minority-Friendly." Communications of the ACM 49(2).

Vilner, T. and Zur, E. 2006. Once She Makes it, She is There: Gender Differences in Computer Science Study. ITiCSE 06: Proceedings of the 11th annual conference on Innovation and technology in computer science education, Bologna, Italy.

Werner, L. L., Hanks, B. and McDowell, C. 2004. "Pair Programming Helps Female Computer Science Students." Journal on Educational Resources in Computing (JERIC) 4(1).

Williams, L., Wiebe, E., Yang, K., Ferzli, M. and Miller, C. 2002. "In Support of Pair Programming in the Introductory Computer Science Course." Computer Science Education 12(3): 197-212.

Wilson, J. D., Hoskin, N. and Nosek, J. T. 1993. The Benefits of Collaboration for Student Programmers. 24th SIGCSE Technical Symposium on Computer Science Education SIGCSE 1993, Indianapolis, Indiana US, ACM Press.

Yin, R. K. 2003. Case Study Research Design and Methods. California USA, Sage Publications Inc.

Experiences in Teaching Quality Attribute Scenarios

Ewan Tempero
Department of Computer Science
University of Auckland
Auckland, New Zealand
ewan-at-cs.auckland.ac.nz

Abstract

The concept of the *quality attribute scenario* was introduced in 2003 to support the development of software architectures. This concept is useful because it provides an operational means to represent the quality requirements of a system. It also provides a more concrete basis with which to teach software architecture. Teaching this concept however has some unexpected issues. In this paper, I present my experiences of teaching quality attribute scenarios and outline Bus Tracker, a case study I have developed to support my teaching.

1 Introduction

It has long been understood that choice of architecture can affect the success of a system (Garlan et al. 1995, Shaw & Garlan 1996). There has been much work done to find ways to efficiently find the right architecture for a given set of requirements. One particular issue is how to establish that a proposed architecture does indeed match the requirements. Doing so requires that the relevant requirements be expressed in an *operational* form that allows an objective assessment of the fit of a given architecture. In the second edition of their book on software architecture, Bass et al. introduced the concept of the *quality attribute scenario* (QAS) (Bass et al. 2003). This concept can be used to express the (non-functional) quality requirements of a system in an operational form.

When I encountered the QAS concept, I thought it was a solution to a problem I faced. In the previous year, I had taught a section on software architecture, and have been unsatisfied with the answers I could give to the question "How do we know when we've got the right architecture?" When I discovered QAS, I immediately saw the potential for providing an objective means to answering the question. There was still the question of how to find the right architecture, but at least having established the QASs for a system we would know what the target looked like. What I didn't anticipate, however, was how difficult a concept it was to grasp for inexperienced software engineers. I speculate that it was easier for me as I had some understanding of what the issues were and so perhaps could more easily see how QAS would help. Prototypical software engineers however are mostly unaware of the concept of software architecture itself, never mind the issues surrounding the development of one. Nevertheless I still had to teach the concept to them!

In this paper, I discuss the problems that students experienced in learning about QAS and outline the **Bus Tracker** case study I have developed and use to support my teaching of software architecture, and QAS in particular.

The rest of this paper is organised as follows. In the next section I will describe the context in which I teach software architecture, so that others can determine how applicable my experience may be to their own situation. In section 3 I will briefly present the quality attribute scenario concept and discuss other related work. Section 4 presents the Bus Tracker case study. Section 5 presents the issues that I have observed students having when learning about QAS, giving examples from the class. Section 6 completes the Bus Tracker case study by providing a set of scenarios for it. I make some concluding comments in section 7.

2 Context

The material described in this paper is taught as part of a Software Engineering (SE) specialisation in the Bachelor of Engineering (BE) degree offered by the School of Engineering at the University of Auckland.

The BE degree is a four year undergraduate programme. Students who are accepted into the programme do a common first year consisting roughly of courses representing all the specialisations in the degree. At the end of the first year, students then apply to enter their specialisation of choice. Each specialisation then consists of 3 years of courses specific to that programme, although each year also has "professional development" courses covering such topics as communication skills, engineering management, ethics, and sustainability.

The SE programme has a course titled "Software Architecture" in the second semester of the second specialisation year (third year of the BE). At the point the students take this course, they would have done one course during the common first year that introduces fundamental programming concepts and a year and half (that is, 3 semesters) of (mostly) SE speciality courses. This includes, during their second year, courses covering topics such as object-oriented design, design patterns, data structures, computer organisation, quality assurance, discrete mathematics, statistics, probability, and a project course. During the first semester of their third year, SE students have courses on databases, human-computer interaction, and computer architecture. While they are taking software architecture they are also taking courses on networks and operating systems, as well as a project course. So the students have seen a number of related topics, however they have not really seen anything where the software architecture of anything is discussed in detail.

The "Software Architecture" course is logically divided into two parts. The first part is "middleware", covering such topics as remote procedure call and replication strategies and technologies such as RMI and CORBA. The second part, which I teach, is software architecture fundamentals. The organisation has come about due to curriculum realities and pedagogical decisions.

We, along with all other curriculum designers, have discovered that 4 years is not enough to teach everything

Copyright ©2009, Australian Computer Society, Inc. This paper appeared at the Eleventh Australasian Computing Education Conference (ACE2009), Wellington, New Zealand, January 2009. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 95, Margaret Hamilton and Tony Clear, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

that we think is important. We have had to make compromises based on our teaching environment, available expertise, local demand, and the usual vocal individuals (Gruba et al. 2004), meaning we could not have two separate courses on middleware and software architecture.

Combining these two areas into one course is not unreasonable as there is a clear relationship between the two. In fact combining middleware with software architecture helped with a problem I had encountered when first teaching this course. Students in their third year of university generally have had little experience with large software systems and so have difficulty appreciating discussions on the architecture of systems. They have also had little experience with quality attributes such as reliability and availability, both of which are important to middleware discussions. By following on from the middleware part, I have some nice examples on which to based discussion on software architecture. The support provided by middleware on teaching software architecture has been noted by others (Royce et al. 1994).

While combining with middleware was useful, it still left me with only 6 weeks in which to cram (what I considered to be) the important fundamental concepts of software architecture. I had originally used the first edition of Bass, Clements, and Kazman's *Software Architecture in Practice*, but as I indicated in the introduction, was unhappy with my ability to answer what I considered to be an important question of any architecture, namely "is it the right one?" When the second edition came out (Bass et al. 2003) and introduced the *quality attribute scenario* concept, I saw it as the answer to my problem. I immediately adopted it and structured the rest of the material around it.

3 Background and Related Work

In this section I give a summary of the concepts needed for the rest of the presentation, and discuss related work. Most of the specifics here come from Bass et al. (Bass et al. 2003)

3.1 Software Architecture Concepts

As many have observed, there are a number of definitions for software architecture (SEI 2007). The specific definition I use in the course is not so relevant to this paper, but for completeness sake it is the one from Bass et al., namely:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

What is not obvious from this definition, although it is made clear in the text, is that what dictates a given architecture is not the functional requirements of a system, but what are often referred to as the non-functional requirements, that is, such things as performance, reliability, extensibility, and so on. These are what are now generally referred to as *quality attributes*. Since these impact choice of architecture, we need to be able to specify them in an operational way in order to be able to determine if the correct choice of architecture has been made. It is to do this that Bass et al. introduce *quality attribute scenarios* (QAS).

Two key points about quality attribute scenarios is that they are intended to capture *requirements*, and so it is not appropriate that they contain any kind of decision regarding how the system may be built, and that they are intended to be used to determine if a given architecture meets those requirements. The second point dictates what kind of information is needed in their description whereas the first dictates what kind of information should not be in their description.

Source	some entity that generates a stimulus
Stimulus	a condition or event that needs to be considered
Artifact	the thing that is stimulated
Environment	conditions under which the stimulus occurs
Response	what the artifact should do on arrival of the stimulus
Measure	how to measure the response to determine it is satisfactory

Figure 1: Quality Attribute Scenario parts (*From Bass et al.*)

Bass et al. identify six "system" quality attributes relevant to software architecture: performance, modifiability, availability, security, usability, and testability. The other commonly discussed quality attributes can be expressed in terms of these six (for example, portability is a specialised form of modifiability, and data integrity is a specialised form of security). There are other aspects that may affect choice of architecture, such as time to market or system lifetime, but it is these six that are my main focus for the use of QASs.

A quality attribute scenario consists of the parts shown in figure 1. It is the "measure" that makes a scenario operational. It is not enough to declare that a web server must be "fast" as we don't know what "fast" means. To some people this may mean generating a response in 100 milliseconds, but others may be content with a response in under 2 seconds.

The "measure" part dictates what is acceptable. But just having the response time is not enough information to evaluate an architecture. For example, does the 2 seconds apply to how long the server has to generate a response from when the request is received, or does it apply to the time between when the user clicks the submit button and sees a page displayed — these two situations involve different "source" and "response" values.

The time the user has to wait for a response depends on factors such as the network performance — the system can hardly be held accountable if the network fails. The context in which the response measure applies is described in the "environment" part.

There can be potentially many scenarios for a given system and so a concern is that some may be missed. Bass et al. distinguish between *general* and *concrete* QASs. General QASs are system independent; the same scenario can apply to many different systems. Bass et al. provide initial lists of possible values for each of the parts of a QAS for each of the system quality attributes. The process of finding QASs begins by choosing relevant values from these lists to identify the appropriate general scenarios. Then each general scenario is instantiated for the particular system under consideration by deciding on system-specific values for each of the general values to get concrete scenarios. One general QAS may result in many concrete QASs. Figure 2 shows the possible values for Performance general scenarios as described by Bass et al.

While I present the general scenario concept, and insist that students at least indicate which values they are using when developing concrete scenarios, I do not regard this concept as a complete solution. So, for example, I make it clear that we should not regard that all the general values provided Bass et al. are a complete set, or even that we must pick exactly one value from each list. General scenarios are (merely) a tool that help us produce concrete scenarios.

3.2 Related Work

Shaw and Clements have argued that software architecture is in its "golden age" and in the near future will reach

source of stimulus	an independent source (possibly within the system)
stimulus	periodic events arrive; sporadic events arrive; stochastic events arrive
environment	normal mode; overload mode;
artifact	system
response	processes stimuli; changes level of service
response measure	latency, deadline, throughput, jitter, miss rate, data loss

Figure 2: Possible general values for Performance (*From Bass et al.*)

the point of being an "unexceptional, essential part of software system building — taken for granted, employed without fanfare, and assumed as a natural base for further progress" (Shaw & Clements 2006). I would argue that to attain this status it needs to be part of any software engineering curricula (and possibly computer science curricula as well). As I have already suggested, and as others have confirmed, there are issues in teaching software architecture, especially to inexperienced undergraduate students.

To my knowledge, there has been no discussion of the teaching of quality attribute scenarios, but there have been some discussions relating to teaching software architecture in general. One of the first was by Royce et al. who discussed the use of a middleware product to teach software architecture at a graduate level (Royce et al. 1994). Their proposed course emphasised developing large-scale systems from reliable, pre-integrated, reusable components, and considered the ability to compare architectures. At the time of publication the course had only been offered once but the authors observed that it was complicated teaching software architecture concepts to students with little practical experience.

In contrast, Bucci et al. explained how they introduced the concept of software architecture early in the curriculum (Bucci et al. 1998). They focused on the view of software presented to developers (or students) by the tools that are used, and argued that tools that provided an architecture-level view of software would help students understand software architecture. While the part of the course I teach involves no actual writing of code, Bucci et al.'s point that students need the right *mental models* is, I believe, at the heart of the teaching software architecture.

More recently Lago and van Vliet discussed their experiences teaching two software architecture courses at the Masters level (Lago & van Vliet 2005). Their course goals included generating alternative architectures, describing architectures, and evaluating architectures, and they were particularly interested in the need to trade-off different stakeholder requirements and consequently the need to communicate effectively with stakeholders. I see the use of QASs as a key ingredient in such communication, as well as evaluating the final result. Lago and van Vliet do not mention QAS explicitly, although they use Bass et al. as the text for one of the courses. They do note the importance of the quality of "scenarios," but it is not clear if they mean QASs specifically, or a more general use of the term.

Karam et al describe their undergraduate presentation of software architecture taught at about the same level as my course (Karam et al. 2004). They present many of the same topics that I do, although they do no explicitly mention QAS. They claim that having complete executable examples allow the students to understand the material better. I am not convinced. As I will discuss later, I have found students tend to want to dive into implementation as soon as possible, even when they are trying to determine requirements. I am concerned that working with executable systems would reinforce that behaviour so my preference is to not deal with actual code.

4 Bus Tracker

The example I use in the case study was initially set as an assignment (see section 4.1). It involves development of a system for providing electronic display of estimated arrival times of public buses for a city council

4.1 History

This example was used as an assignment in the first offering of the course in 2002. The assignment was to "develop an architecture for the system." It was not used in 2003 (when QASs were introduced) but then reused in 2004 as the basis for the two-assignment sequence (see section 4.4). It worked so well for the assignment that in 2005 I decided to begin developing it as a case study for use in class and tutorials. The case study was then developed over the next 3 deliveries of the course.

4.2 Description

Figure 3 shows some of the initial details that are given to the students (the explanation at the bottom has been added for this presentation). Some other details are also provided, partly to provide some more concreteness to the exercise. For example, some of the functionality is described in more detail, such as what should appear on a display. Numbers are also given for how many bus stops and buses may need to be considered, and some possible performance characteristics of the communications systems and other hardware.

Nevertheless there are still many details that might be important to know when developing an architecture. For example, nothing is said about what hardware is available (some Computer Systems Engineers take the course each year and they have observed that different chipsets could be used for the bus subsystems with different capabilities and different costs), or how the estimates are produced. Partly this is due to my lack of knowledge in such things, and anyway such details also should not be so relevant to developing the quality requirements. But I use this lack of information to make the point that as software architects the students are likely to be in the same situation (not having complete information) and they are welcome to make up any details they feel are necessary so long as they can justify their choices and the choices do not make the exercise trivial.

In fact, the details I do give them are not totally consistent, or not that relevant, or not actually likely to be what the client actually wants, as I will discuss later. Over the years I have resisted the temptation to "improve" the information. Information provided by clients is notorious for not necessarily being of high quality and others have observed that development of the architecture is sometimes where the true requirements are determined (Bass et al. 2003, sidebar, p27). This is a point I can make more easily using the existing description than if I had provided a "sanitised" description.

Another aspect of this example that has proven useful is that it naturally decomposes into three subsystems: the bus subsystem, the display subsystem, and the rest (which I typically refer to as the "central" subsystem, although there is nothing implied in the description that that part has to be all in one place). This provides a nice example of how the system architecture can partially impact the choice of software architecture, as well as the distinction between system and software architecture.

As well as the aspects I've mentioned above, the Bus Tracker system is useful for the number of architectural questions that arise. In particular, there are are multiple examples of all the quality attributes, all leading to many interesting different possible scenarios. I will discuss some of these in section 6.

ARC wants a system called Bus Tracker that tracks buses. It wants to add GPS to all of its buses so that it can track where they are to within 100 metres. They will use this information to provide estimated arrival times of buses at each major bus stop.
The unit to be placed on each bus consists of a Global Positioning System (GPS) receiver, a radio transmitter, and other bits of hardware and software. The GPS receiver can can determine its position to within 10 metres at each second. If calibrated properly, it can reliably track the bus position and speed throughout the journey. It transmits this information, along with the bus' identifier to the Bus Tracker system on a regular basis.
The major bus stops are where a number of bus routes converge. There are typically 20 or more buses that stop at these stops during peak travel times. The planned displays will have a radio receiver and room for display four or five bus numbers and times (that is, similar to those that already exist for the LINK bus).
The displays should repeatedly scroll through all the buses whose estimated or scheduled arrival times at that stop are sometime in the next hour. Once the bus is within 1 kilometre (that is, about 2 bus stops away) of a display, the estimated arrival time should be within 2 minutes of the actual time, 95% the time. All other displays should show a "best effort" estimated time.
The bus company also would like to allow bus users to get estimated arrival times for all buses at all times via their web site, and also via phone.

(The ARC — Auckland Regional Council — is an elected local government authority covering the Auckland Region and the cities within it with regulatory power and funding capabilities for such things as public transport, environmental protection and regional parks. The LINK bus is a particular bus service.)

Figure 3: Bus Tracker initial description.

4.3 Presentation

The way the case study is presented has evolved over the last 3 or so years. What is described here is the presentation used in the 2007 second semester (July-October) offering.

The first step is a tutorial session in which the students work in teams of about 5 members (self-formed) to identify the "non-functional" requirements likely to be relevant to the text as given. The purpose of this tutorial is to demonstrate the issues with specifying such requirements in a manner that allows for proposed architectures to be evaluated. It also has the benefit of giving the students a chance to come to grips with the Bus Tracker system itself, without the distraction of having to apply new concepts at the same time. Teams (the number depending on the amount of time available) are then asked to present one non-functional requirement to the class, and we discuss how useful their description is with respect to being able to determine if a proposed architecture meets the requirement.

At the time of this tutorial, the students have not seen the QAS concept, but they have had explained to them the general idea of what software architecture means and why architecture is important, and have seen fairly high-level descriptions of some software architecture examples.

The second step is a tutorial a week later. By this time the students have seen QAS, including general scenarios. In this tutorial, they are formed into teams (this time not of their choosing) and asked to develop two performance scenarios for Bus Trucker. Once all the teams have at least one completed (typically after about 30 minutes), one team is picked to present one of their scenarios to the rest of the class. This scenario is then critiqued.

Later on the same day, the course has a scheduled lab. We use this lab session to refine the scenario descriptions, with each team entering their descriptions into the SE Wiki.

In the next lecture session we go through several submitted scenarios for another round of feedback. Finally, in a following lecture, I present some scenarios of my own and explain my reasoning for choosing (or rejecting) them.

Although not directly relevant to this paper, the Bus Tracker system is later used as the basis for tutorials, and lab exercises on the development of structures for architectures.

4.4 Assignments

It is worth noting that the assignments follow the same pattern. Initially, students given a piece of text at a similar level of detail as in figure 3. Their first assignment is then to develop some number (typically 3) of quality attribute scenarios for a quality attribute (typically performance or availability) for the proposed system, and also one structure description that relates to one of their scenarios. These assignments are marked via peer assessment using an on-line peer assessment system (Hamer et al. 2007).

For the second assignment, I give them 2-3 scenarios and ask them to develop an architecture that meets those requirements. Doing this means that they get to see another set of scenarios I have developed for another system that they have had to think about, reinforcing the sequence from Bus Tracker. They submit their architecture description and justify it with respect to the scenarios. As this assignment is due only at the end of the teaching period, it is assessed in the more traditional fashion.

5 Issues

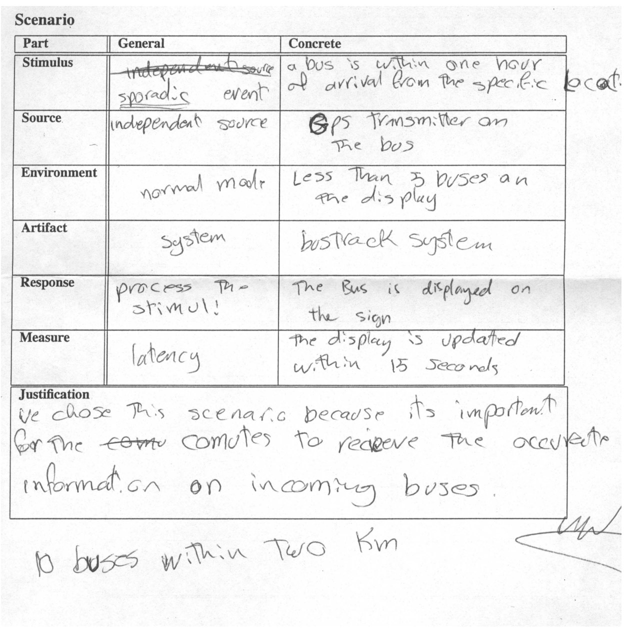
In this section I discuss the kinds of issues I have observed.

5.1 Overview

There are four areas of confusion that need to be addressed — what constitutes reasonable values for each part, how the values for each part interact, whether the choice of values constitutes a quality requirement, and whether the resulting scenario is relevant to the system being developed. Teaching QASs requires progressing through each of these areas. The first two areas are about how to construct a valid scenario, whereas the last two are about whether the scenarios are describing something useful.

Determining reasonable values is about such things as describing something for the stimulus part that really is a stimulus, or an artifact that really is a part of the system that is relevant to the creation of the architecture. The first scenarios produced by the students usually contain values that are somewhat acceptable, but lack the precision needed for the ultimate use of QASs — evaluating an architecture. However in some cases the values are not acceptable. A common example is the statement for the measure that contains nothing measurable

While it might be expected that the general scenario values would help reduce these problems, my experience is that early on, students have few problems creating consistent general scenarios but have difficulty with concrete scenarios. I speculate that this is because the production of a general scenario can be done by just choosing values from lists. This means a consistent general scenario can be produced without a great deal of understanding of what the individual values mean. A common mistake is to produce an acceptable general scenario, but then choose values for the concrete scenario that are inconsistent with the



Scenario

Part	General	Concrete
Stimulus	~~independent source~~ sporadic event	a bus is within one hour of arrival from the specific locat.
Source	independent source	GPS Transmitter on The bus
Environment	normal mode	Less Than 5 buses an the display
Artifact	system	bustrack system
Response	process The stimuli	The Bus is displayed on the sign
Measure	latency	the display is updated within 15 seconds

Justification

we chose This scenario because its important for The ~~comu~~ comutes to recieve The accurate information on incoming buses.

10 buses within Two Km

Figure 4: A typical first QAS. (Most details are reproduced in the main text.)

chosen general values. For example, choosing "latency" as the general performance measure, but then specifying a concrete measure that is not latency.

The next area is choosing values that work together to describe a scenario. Here the issue is whether, for example, the stated stimulus can be produced by the stated source, and will be felt by the stated artifact, or whether the stated measure does measure something related to the stated response. It is common to see sets of values where the individual values make sense, but they do not fit together to make a coherent QAS.

Once a valid scenario is constructed, there is the question as to whether it is of any use. One property that reduces the usefulness of scenarios is when it assumes or dictates architectural decisions. It is common early on for students to, effectively, think about how they would build the system they are supposed to be developing QASs for, and then write the scenarios with their designs in mind. While it is possible that some aspects of an architecture may be dictated by the client ("must use J2EE" or even "must use replication to ensure availability") it is not appropriate for the architect to add architectural details not already given in the requirements.

Even once we have valid scenarios that really do specify requirements, there is still the question as to whether or not they specify the right requirements, that is, those intended by the client. While this is a crucial property of a useful scenario — if we get it wrong then the wrong architecture may result — it is in some sense the one I'm least concerned about. If the students can produce scenarios that a client can immediately determine are specifying the wrong things, then I have done my job. If the QASs have been produced to a level of quality that clients can, with confidence, figure out that they are the wrong requirements, then at least the (big) problem of mis-communication has been reduced.

5.2 Examples

I will now illustrate the comments above with examples.

In the first QAS exercise, the students are asked to produce performance scenarios for the Bus Tracker system. As well as producing the concrete scenario, the students are also required to give the general scenario (ideally starting with that), and also explain why they have picked the scenario they have in terms of the details of the system they have been given. The latter requirement is intended to prevent students choosing a scenario they think *should* exist, as opposed to one based on the information provided by the client. This is to remind students that once they become professional engineers, their responsibility is to their client, and so they are not free to just make up stuff they think might be interesting. That said, as I mentioned earlier, I don't provide complete details and so they do sometimes have to fill in the gaps. My main requirement is that whatever assumptions they make are not inconsistent (note the deliberate use of the double negative) with the text I give.

Figure 4 shows a typical first attempt at a performance QAS for the Bus Tracker system (in fact produced by one of the teams in the 2007 class). The first point to make about this scenario is that the team has identified a reasonable requirement for the Bus-Tracker system, namely that providing the estimated arrival times on the displays in a timely manner is a key requirement. The second point to make is that the general scenario details are reasonable for the requirement they are trying to specify. This suggests that any problems the students have is not due to lack of understanding as to what they are trying to do, or the general idea of what a scenario looks like. The problems are with their choice of values for the concrete scenario, both for the specific parts, and for the overall scenario.

The first problem is with the stated stimulus: "a bus is within one hour of arrival from the specific location." This does not describe something that might be regarded as an actual stimulus. My experience is that the wording is indicative of confusion as to what constitutes a stimulus, and this is an issue that needs further discussion and explanation.

A generous interpretation of what has been written might assume that it was intended to be something like "a bus reaches a point that is one hour from the specific location," which is closer to being a stimulus, however it is a somewhat nebulous statement, and, more significantly is inconsistent with both the stated source "gps transmitter on the bus" and the stated artifact "bustrack system" — the transmitter doesn't cause a bus to arrive anywhere, and the bus' arrival at some point doesn't directly have any effect on the overall system. Note that in this case it is not the values of source and artifact that are the problem, but the relationships between values for different parts of the scenario.

A stimulus value that would be more consistent with the stated source and artifact would be something like "a message from a bus within one hour from the location is received." This suggests that there needs to be another scenario for when the buses are further away. As QAS are intended to capture quality requirements, there should only be separate scenarios if the quality requirements are different. In this case, it might be reasonable to suppose that the display does not have to be updated in quite so timely a manner.

The choice of source indicates some confusion. For a start, a GPS system typically doesn't transmit anything, but that could be due to either lack of knowledge about how these things work, or what was written being shorthand for "The subsystem on the bus containing the GPS receiver." Nevertheless it is representative of a lack of precision regarding what exactly constitutes the values of the different parts of the scenarios.

The choice of artifact also shows a lack of precision. In the context given, we can probably reasonably assume that it is the "central" system that is meant, but, as with any presentation of requirements, we would prefer not to have to assume anything.

While beginners generally seem to understand what most of the parts are supposed to do (even if they struggle to apply this understanding to produce sensible values), the environment part usually causes the most confusion in terms of its purpose. The value given in the example "Less than 5 buses on the display" is, on the face of it, not an unreasonable value. However it implies that there must be a different scenario for the case when there are more than 5 buses (needed to be) on the display. While it is likely that in terms of *implementation* these two cases may be different, it seems unlikely that the quality requirements for the two cases should be different, and so separate scenarios are unnecessary. I believe this confusion is due to the students still thinking in terms of what they have to do to produce the system, rather than thinking about what the quality requirements of the system are. The fact that scenarios are intended to specify *requirements* (not design or implementation details) and in particular *quality* requirements is in my experience something that needs to be repeated and reinforced.

One point to make about this example is, once the issues regarding the other parts of the scenario are resolved, the stated measure, "The display is updated within 15 seconds," and response, "The Bus is displayed on the sign," are reasonable values. It is not uncommon to get measures such as "before the bus gets to the next stop", which, as well as not really being a measurement, is not something that any proposed architecture could be evaluated against.

Once all the issues discussed above are resolved, there is still the question as to how useful the resulting scenario is. It may describe something that looks like a performance requirement, but is it one that we would care about when developing an architecture? What this scenario does not do is specify which display is being updated. A bus does not go to just one bus stop, and most of the time it will be within one hour of several bus stops, with more than one having displays. It is not sufficient that the system update on bus stop in a timely manner, but that it update several. The need to generate multiple estimates and deliver them to multiple displays is likely to be significant in terms of performance requirements. This is a point that is always missed.

Related to the above point is the fact that there are also multiple buses, including multiple buses on the same route (and so visiting the same displays in the same order). This point is not directly evident in the given scenario. However, it seems clear that the team was somewhat aware of this due to their choice of "sporadic event" for the general value of the scenario, which only makes sense if there are multiple (albeit unpredictably spaced) events.

There is one remaining problem with the example scenario, which I will address in section 6.

5.3 Other comments

As I mentioned earlier there was usually little difficulty in coming up with a consistent general scenario, but there are problems deciding on concrete values to match the general values. In part, some of those problems are due confusion as to what the general values meant. For example, distinction between "sporadic" and "stochastic" was often unclear, as are the differences between "latency", "throughput", and "deadline." Even when it was clear that these terms had been encountered before, there appeared to be difficulty applying them in this context. This showed up in performance discussions most often, but that is almost certainly because that was the quality attribute I used the most, figuring students would have better intuition about it than the less familiar quality attributes.

6 Bus Tracker Scenarios

In this section I will present some of the scenarios I use in the course to illustrate various points about QAS development.

The starting point is to reexamine the information that has been provided. While I make the point that we have a responsibility to the client, that doesn't mean we should simply accept what we are given uncritically! Close examination of the given text reveals some problems. For example, consider the phrase "*the estimated arrival time should be within 2 minutes of the actual time, 95% of the time*." While it seems reasonable that the estimate should be reasonably accurate (otherwise what is the point of having it), it is not clear what "95% of the time" means. If this is determined over a day's operation, which is from 6am to 12am, then if the system is down for 1 hour it will not meet this requirement. However, 1 hour of downtime over a week does meet this requirement.

Another problem is the phrase "*Once the bus is within 1 kilometre*" which is intended to indicate that if the bus is a reasonable distance away then the estimates can be less accurate. However, it takes 3 minutes to travel 1 kilometre at 20 km/h. Having a 2 minute accuracy requirement when the bus is only about that far away seems fairly pointless! Notice that these issues didn't cause the problems the students faced in developing their scenarios.

Instead, we need to determine what the client is really trying to say. One reasonable interpretation is:

> *Provide estimated times of arrival accurate enough to be useful.*

Now we can start considering the quality implications of this requirement. Examples include:

- can the system "keep up," that is show estimates in time to be useful? — *performance*
- can we add new displays quickly and/or cheaply while still showing accurate enough estimates in time to be useful? — *modifiability*
- under what conditions must the system show accurate enough estimates in time to be useful? — *availability*

Starting with performance, we need to express what it means for the system to "keep up" or, "show accurate enough estimated times of arrivals for all buses on all displays". A question that is always raised at this point is concern of how to specify this requirement without knowing how to get accurate enough estimates, or whether the estimates can be produced fast enough. There are two answers I give to this question. The first is that, while as engineers we are responsible for building the system, that does not mean we have to build every little bit of it ourselves. For such things as estimation algorithms, we could contract that out to experts (e.g., those with a more traditional computer science background). The second answer is that that question isn't actually relevant when trying to come up with QASs. The client determines how accurate the estimate has to be and that dictates how quickly the estimates have to be generated. It is up to us, once we know what "quickly" means, to find develop an architecture that will meet the requirements (or convince the client to accept something a little less expensive). For the moment, the question that needs to be answered is what "quickly" means.

Source	A bus subsystem
Stimulus	. . . sends out a message with its current speed and location every 15 seconds
Artifact	. . . to the central subsystem
Environment	. . . when all communications and hardware is working adequately.
Response	The system produces an estimated arrival time for all relevant displays and sends them out to the display where the estimate is shown
Measure	. . . in under 30 seconds from when the bus' message was sent.

Figure 5: Performance Scenario A

Figure 5 gives my version of a scenario like that shown in the previous section. It has an issue that the students are quick to point out — how does it make sense to have 15 seconds in one place and 30 in an other? Note that this is not a problem with the integrity of the scenario itself, but is a perceived problem with the requirements it is supposed to capture. This is what is good about QASs. They allow for a discussion about what the precise requirements are, rather than having to guess.

Source	A bus subsystem
Stimulus	. . . sends out a message with its current speed and location every **60** seconds
Artifact	. . . to the central subsystem
Environment	. . . when all communications and hardware is working adequately.
Response	The system produces an estimated arrival time for all relevant displays and sends them out to the display where the estimate is shown
Measure	. . . in under **15** seconds from when the bus' message was sent.

Figure 6: Performance Scenario B

Figure 6 shows a scenario that fixes the problem with Scenario A. It says that each bus sends out its speed and location every 60 seconds, and all updated estimates based on that information must be shown within 15 seconds of the message being sent. This is, however, a more subtle problem with this scenario — it suggests that a new estimate be delivered to every relevant display every time every bus sends a message. In fact, there is nothing in the requirements implied by this scenario that prevents an implementation just sending an old estimate (if it's not too old). Nevertheless the scenario does seem to be specifying more than is intended.

Source	A bus subsystem
Stimulus	. . . sends out a message with its current speed and location every 5 seconds
Artifact	. . . to the central subsystem
Environment	. . . when all communications and hardware is working adequately.
Response	The system stores the bus' id, location, and speed with a timestamp for when the message was received
Measure	. . . in under 1 second from when the message is received.

Figure 7: Performance Scenario C

Figure 7 shows a better scenario, in that all it specifies is that messages that are sent by buses are quickly recorded. But a feature of all of these scenarios is that they are "internal", in that the stimuli are all generated from within the system, and they imply part of the architecture (the three subsystems and to some extent how they interact). The latter point should especially be of concern. For example, all three scenarios assume that the bus subsystems "push" their information to the rest of the system, rather than the central system polling buses for their current speed and location. While the push style interaction probably makes most sense, requirements that imply design are to be avoided.

Figure 8 shows a scenario that I feel is more useful for describing some of the performance requirements for Bus Tracker. The stimulus is external to the system, namely a user of the system wanting to use its functionality, and all it says is that what is shown on the display should be based on fairly current information about the relevant bus' speed and location. Note the level of detail provided in the environment part. This performance requirement does not apply if there are more than 20 buses scheduled to arrive (so, so long as the display can scroll through 20 buses in 30 seconds the patron will see her bus' time within 30 seconds), nor does it apply if the bus is estimated to be more

Source	A bus patron
Stimulus	. . . wants to know when their bus is going to arrive at
Artifact	. . . the bus stop they're standing at when
Environment	. . . the desired bus is estimated to arrive in less than 20 minutes and at most 20 buses are estimated to arrive within 20 minutes of the bus stop.
Response	The display at the bus top shows an estimated time of arrival of the desired bus based on the actual location of the bus that is not more than 1 minute old
Measure	. . . in under 30 seconds.

Figure 8: Performance Scenario D

than 20 minutes away (although presumably the estimated time will *eventually* appear on the display, just not in 30 seconds — another scenario would be needed to specify that).

Now consider the modifiability requirement. Adding a new display can be a non-trivial operation. The equipment needs to be assembled, a hole dug and wires connected. If we were responsible for building the entire system, we might need to make decisions regarding this aspect, but in terms of developing a *software* architecture, it's the software modifications we need to consider. The kinds of questions we need to think about includes such things as: does the change have to be possible when the system is operational or can we shut the system down, how do we measure the cost of modification — time, money, or something else.

Source	Operations manager
Stimulus	. . . requests that a new display be made operational
Artifact	. . . in the Bus Tracker system
Environment	. . . outside of the peak traffic period.
Response	The new display starts displaying estimated arrival times for buses relevant to it
Measure	. . . within 5 minutes of the request

Figure 9: Modifiability Scenario E

Figure 9 shows the scenario that requires that a new display must be able to be brought on-line without affecting the rest of the system, but only during a time that is outside peak traffic (perhaps because the client does not want to do anything that will affect the system's behaviour during the time when most of the patrons want to use it). This scenario assumes that all the equipment is already in place.

Source	Operations manager
Stimulus	. . . requests that a new display be made operational
Artifact	. . . in the Bus Tracker system
Environment	. . . **while no buses are running**
Response	The new display is available to start displaying estimated arrival times for buses relevant to it
Measure	. . . **within 4 hours**.

Figure 10: Modifiability Scenario F

Figure 10 shows another possibility. If the client agreed that this scenario was acceptable then it would allow the system to be shut down and restarted, which may allow a choice of architecture (and so system) that is cheaper than what would be required to meet Scenario E.

For the availability requirements, we need to consider what constitutes "useful", for example this might mean

Source	A random event
Stimulus	...causes a failure
Artifact	...to the subsystem on a bus
Environment	...while the bus is on a bus route.
Response	The relevant displays must start showing the scheduled arrival time for the bus
Measure	...until it next starts a route

Figure 11: Availability Scenario G

"most" of the time estimated arrival times for any given bus must be shown, and the remainder of the time it's acceptable to show the scheduled arrival time. Figures 11 and 12 show two possibilities. Scenario G represents the expectation that the failure of a single bus' subsystem should not impact the rest of the system and be fixable once the bus returns to base (that is, fast enough that it can be done before the next time the bus goes out on a route). Note that this measure for this scenario is not really a useful one as presented. We really need specific details as to how long bus routes are and how quickly buses get turned around to go out on another route.

Source	A random event
Stimulus	...causes a failure
Artifact	...to the communications network
Environment	...during normal operation.
Response	All displays must start showing the scheduled arrival time for all buses
Measure	...within 30 seconds of the failure.

Figure 12: Availability Scenario H

Scenario H indicates what has to happen when the communications network goes down. One possibility could have been to just show nothing in this case, but this scenario requires that all the displays showed the scheduled arrival times for all buses. This requirement has an architectural implication. It implies that all displays must have access to these scheduled times, which means that they will need local copies of those times.

7 Concluding Comments

I have described some of the issues I have encountered in teaching the concept of Quality Attribute Scenarios to third year software engineering students. I have taught software architecture for 6 years now, 5 of which using QAS. I have presented some of the details of the Bus Tracker case study to aid my teaching of this concept.

My goal in teaching software architecture is not to produce software architects — as with any other engineering discipline the theory has to be tempered with real-world experience before it becomes useful. Many have suggested that to teach software architecture (and software engineering in general) the coursework needs to be as "real world" as possible. This is, however, difficult in a university environment, and I think we must be content with giving as much practical application of the theory as possible. My conclusion based on my experiences teaching this course is that, at least at the level I am teaching, the students don't have the necessary experience to appreciate a "real world" example. They are still struggling to understand the fundamental principles, and if these are not properly understood, no amount of real world examples will help.

In the case of QASs, practical application takes the form of multiple cycles of creating scenarios and evaluating them for multiple applications. My goal is to make sure that the theory is understood. Based on the assessment items relating to QASs (assignment and typically an exam question) I am meeting that goal, at least with respect to the QAS concept.

Acknowledgements

My thanks to the students I have taught in the 6 years of software architecture teaching I have done. They asked the hard questions that exposed the issues. In particular, thanks to the 2007 Part III software engineering cohort, whose examples have been used here with their permission.

References

Bass, L., Clements, P. & Kazman, R. (2003), *Software Architecture in Practice*, 2 edn, Addison-Wesley.

Bucci, P., Long, T. J. & Weide, B. W. (1998), Teaching software architecture principles in CS1/CS2, *in* 'ISAW '98: Proceedings of the third International Workshop on Software Architecture', ACM Press, New York, NY, USA, pp. 9–12.

Garlan, D., Allen, R. & Ockerbloom, J. (1995), 'Architectural mismatch: Why reuse is so hard', *IEEE Software* **12**(6), 17–26.

Gruba, P., Moffat, A., Sondergaard, H. & Zobel, J. (2004), What drives curriculum change?, *in* R. Lister & A. L. Young, eds, 'Sixth Australasian Computing Education Conference (ACE2004)', Vol. 30 of *CRPIT*, ACS, Dunedin, New Zealand, pp. 109–117.

Hamer, J., Kell, C. & Spence, F. (2007), Peer assessment using Aropä, *in* 'ACE '07: Proceedings of the ninth Australasian conference on Computing education', Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 43–54.

Karam, O., Qian, K. & Diaz-Herrera, J. (2004), A model for SWE course "software architecture and design", *in* '34th Annual Frontiers in Education (FIE)', pp. 4–8.

Lago, P. & van Vliet, H. (2005), Teaching a course on software architecture, *in* '18th Conference on Software Engineering Education and Training (CSEE&T)', pp. 35–42.

Royce, W., Boehm, B. & Druffel, C. (1994), Employing unas technology for software architecture education at the university of southern california, *in* 'WADAS '94: Proceedings of the eleventh annual Washington Ada symposium & summer ACM SIGAda meeting on Ada', ACM Press, New York, NY, USA, pp. 113–121.

SEI (2007), 'SEI list of definitions of software architecture', `http://www.sei.cmu.edu/architecture/definitions.html` Accessed September.

Shaw, M. & Clements, P. (2006), 'The golden age of software architecture', *IEEE Software* **23**(2), 31–39.

Shaw, M. & Garlan, D. (1996), *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.