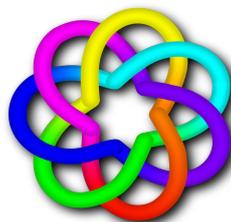


CONFERENCES IN RESEARCH AND PRACTICE IN
INFORMATION TECHNOLOGY

VOLUME 123

COMPUTING EDUCATION 2012

AUSTRALIAN COMPUTER SCIENCE COMMUNICATIONS, VOLUME 34, NUMBER 2



COMPUTING EDUCATION 2012

Proceedings of the
Fourteenth Australasian Computing Education Conference
(ACE2012), Melbourne, Australia,
31 January – 3 February 2012

Michael de Raadt and Angela Carbone, Eds.

Volume 123 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 2012. Proceedings of the Fourteenth Australasian Computing Education Conference (ACE2012), Melbourne, Australia, 31 January – 3 February 2012

Conferences in Research and Practice in Information Technology, Volume 123.

Copyright ©2012, 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:

Michael de Raadt

Moodle Pty Ltd
1/224 Lord St
Perth, WA, 6000
Australia
Email: michaeld@moodle.com[.07in]

Angela Carbone

Office Pro Vice-Chancellor (Learning and Teaching)
Monash University
Caulfield East, VIC, 3145
Australia
Email: angela.carbone@monash.edu

Series Editors:

Vladimir Estivill-Castro, Griffith University, Queensland
Simeon J. Simoff, University of Western Sydney, NSW
Email: crpit@scm.uws.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 123.
ISSN 1445-1336.
ISBN 978-1-921770-04-3.

Printed, January 2012 by University of Western Sydney, on-line proceedings
Printed, January 2012 by RMIT, electronic media
Document engineering by CRPIT

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

Table of Contents

Proceedings of the Fourteenth Australasian Computing Education Conference (ACE2012), Melbourne, Australia, 31 January – 3 February 2012

Preface	vii
Programme Committee	ix
Organising Committee	x
Welcome from the Organising Committee	xi
CORE - Computing Research & Education	xiii
ACSW Conferences and the Australian Computer Science Communications	xiv
ACSW and ACE 2012 Sponsors	xvi

Keynote

The Future of Educational Programming Tools What Will Come (Or At Least Should Come)	3
<i>Michael Kölling</i>	

Contributed Papers

Perceptions of a gender-inclusive curriculum amongst Australian Information and Communications Technology academics	7
<i>Tony Koppi, Madeleine Roberts and Golshah Naghdy</i>	
Attrition from Australian ICT Degrees Why Women Leave	15
<i>Madeleine Roberts, Tanya McGill and Peter Hyland</i>	
Work Integrated Learning Rationale and Practices in Australian Information and Communications Technology Degrees	25
<i>Chris Pilgrim and Tony Koppi</i>	
Trends in Introductory Programming Courses in Australian Universities Languages, Environments and Pedgogy	33
<i>Raina Mason, Graham Cooper and Michael de Raadt</i>	
Teaching Novice Programming Using Goals and Plans in a Visual Notation	43
<i>Minjie Hu, Michael Winikoff and Stephen Crane</i>	
Toward a Shared Understanding of Competency in Programming: An Invitation to the BABELnot Project	53
<i>Raymond Lister, Malcolm Corney, James Curran, Daryl D'Souza, Colin Fidge, Richard Gluga, Margaret Hamilton, James Harland, James Hogan, Judy Kay, Tara Murphy, Mike Roggenkamp, Judy Sheard, Simon and Donna Teague</i>	
Introductory programming: examining the exams	61
<i>Simon, Judy Sheard, Angela Carbone, Donald Chinn, Mikko-Jussi Laakso, Tony Clear, Michael de Raadt, Daryl D'Souza, Raymond Lister, Anne Philpot, James Skene and Geoff Warburton</i>	

Student Created Cheat-Sheets in Examinations: Impact on Student Outcomes	71
<i>Michael de Raadt</i>	
Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions	77
<i>Malcolm Corney, Donna Teague, Alireza Ahadi and Raymond Lister</i>	
Swapping as the “Hello World” of Relational Reasoning: Replications, Reflections and Extensions . . .	87
<i>Donna Teague, Malcolm Corney, Alireza Ahadi and Raymond Lister</i>	
Models and Methods for Computing Education Research	95
<i>Mats Daniels and Arnold Pears</i>	
Illustration of Paradigm Pluralism in Computing Education Research	103
<i>Neena Thota, Anders Berglund and Tony Clear</i>	
Switches CAM Table Poisoning Attack: Hands-on Lab Exercises for Network Security Education	113
<i>Zouheir Trabelsi</i>	
Implementation of smart lab for novice programmers	121
<i>Ali Saleh Alammery, Angela Carbone and Judy Sheard</i>	
Evaluation of an Intelligent Tutoring System used for Teaching RAD in a Database Environment . . .	131
<i>Silviu Risco and James Reye</i>	
Using Quicksand to Improve Debugging Practice in Post-Novice Level Students	141
<i>Joel Fenwick and Peter Sutton</i>	
Coming to terms with Bloom: an online tutorial for teachers of programming fundamentals	147
<i>Richard Gluga, Raymond Lister, Judy Kay, Tim Lever and Sabina Kleitman</i>	
An exploration of factors influencing tertiary IT educators’ pedagogies	157
<i>Sally Firmin, Judy Sheard, Angela Carbone and John Hurst</i>	
Common Areas for Improvement in ICT Units that have Critically Low Student Satisfaction	167
<i>Angela Carbone and Jason Ceddia</i>	
Dimensions and Directions for Strategies to Address IT Students Cheating and Plagiarism Practices .	177
<i>Judy Sheard and Martin Dick</i>	
Why the Bottom 10% Just Can’t Do It - Mental Effort Measures and Implications for Introductory Programming Courses	187
<i>Raina Mason and Graham Cooper</i>	
Author Index	197

Preface

Welcome to the Fourteenth Australasian Computing Education Conference (ACE2012). This year, the ACE2012 conference, which is part of the Australasian Computer Science Week, is being held in Melbourne, Australia from 31 January to 3 February, 2012.

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 21 full papers from the 43 papers submitted, giving an acceptance rate of 49%. The number of submissions was slightly less than the 47 papers submitted in the previous year, however this year we had seven papers submitted by research students, which reflects the growing research interest in computing education. We again see a strong international presence, with submissions from Australia, New Zealand, Japan, Sweden, Finland, United Kingdom, United States, United Arab Emirates, India and Iran.

This year we were successful in bidding for an ACE invited keynote speaker to fill an ACSW plenary session. Michael Kölling, Professor at the School of Computing, University of Kent, in Canterbury, UK will be delivering the keynote address. Michael will also deliver a workshop on *Introductory Programming Teaching with Greenfoot* prior to the commencement of the conference. Due to the excellent response to our Call for Papers and the number of accepted papers this year, we have chosen not to have any invited speakers.

A variety of topics are presented in this year's papers, including: novice programmer education; gender issues, tools; work-integrated learning; computing education research; exam standards and pedagogy. Many of the papers have present new innovations, and many demonstrate high quality research.

As with past ACE conferences, we are continuing to hold workshops at ACE. Five workshops have been organized. Apart from the above mentioned workshop, others include: *Developing an Exam Taxonomy* led by Judy Sheard and supported by the Learning and Teaching Academy created by the Australian Council of Deans in Information Communication Technology; *Road testing the Peer Assisted Teaching Scheme* supported by Australian Learning and Teaching Council (ALTC) Teaching Fellowship led by Angela Carbone; *Improving Teaching: designing and facilitating for learning at the subject level* led by Sue Wright and Jocelyn Armarego and *Epistemology of Competency* led by Ray Lister and supported by ALTC Innovation grant.

For the first time, ACE awarded a best paper and best student paper. This year the best paper was awarded to:

- ★ Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions
Malcolm Corney, Donna Teague, Alireza Ahadi and Raymond Lister

Three other papers were also highly praised during reviews. These commendable papers were:

- ★ Toward a Shared Understanding of Competency in Programming: An Invitation to the BABELnot Project
Raymond Lister, Malcolm Corney, James Curran, Daryl D'Souza, Colin Fidge, Richard Gluga, Margaret Hamilton, James Harland, James Hogan, Judy Kay, Tara Murphy, Mike Roggenkamp, Judy Sheard, Simon and Donna Teague
- ★ Introductory programming: examining the exams
Simon, Judy Sheard, Angela Carbone, Donald Chinn, Mikko-Jussi Laakso, Tony Clear, Michael de Raadt, Daryl D'Souza, Raymond Lister, Anne Philpot, James Skene and Geoff Warburton
- ★ Trends in Introductory Programming Courses in Australian Universities Languages, Environments and Pedagogy
Raina Mason, Graham Cooper and Michael de Raadt

The best student paper was awarded to:

- ★ Trends in Introductory Programming Courses in Australian Universities Languages, Environments and Pedagogy
Raina Mason, Graham Cooper and Michael de Raadt

We are grateful to SIGCSE 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 RMIT, Melbourne, and the Australasian Computing Education executive for the opportunity to chair the ACE2012 conference.

Michael de Raadt
Moodle

Angela Carbone
Monash University

ACE2012 Conference Co-chairs
January 2012

Programme Committee and Additional Referees

Chairs

Michael de Raadt, Moodle
Angela Carbone, Monash University, Australia

Members

David J. Barnes, University of Kent, UK
Tim Bell, University of Canterbury, New Zealand
Alison Clear, AUT University, New Zealand
Tony Clear, AUT University, New Zealand
Nell Dale, University of Texas at Austin, USA
Mats Daniels, Uppsala University, Sweden
Julian Dermoudy, University of Tasmania, Australia
Sally Fincher, University of Kent, UK
John Hamer, University of Auckland, New Zealand
Margaret Hamilton, RMIT University, Australia
Mikko Laakso, University of Turku, Finland
Raymond Lister, University of Technology, Sydney, Australia
Chris McDonald, University of Western Australia, Australia
Arnold Pears, Uppsala University, Sweden
Anne Philpott, AUT University, New Zealand
Helen Purchase, University of Glasgow, UK
Anthony Robins, Otago, New Zealand
Judy Sheard, Monash University, Australia
Simon, University of Newcastle, Australia
Josh Tenenberg, University of Washington, USA
Jacqueline Whalley, AUT University, New Zealand

Conference Webmaster

Michael de Raadt, Moodle

Organising Committee

Members

Dr. Daryl D'Souza
Assoc. Prof. James Harland (Chair)
Dr. Falk Scholer
Dr. John Thangarajah
Assoc. Prof. James Thom
Dr. Jenny Zhang

Welcome from the Organising Committee

On behalf of the Australasian Computer Science Week 2012 (ACSW2012) Organising Committee, we welcome you to this year's event hosted by RMIT University. RMIT is a global university of technology and design and Australia's largest tertiary institution. The University enjoys an international reputation for excellence in practical education and outcome-oriented research. RMIT is a leader in technology, design, global business, communication, global communities, health solutions and urban sustainable futures. RMIT was ranked in the top 100 universities in the world for engineering and technology in the 2011 QS World University Rankings. RMIT has three campuses in Melbourne, Australia, and two in Vietnam, and offers programs through partners in Singapore, Hong Kong, mainland China, Malaysia, India and Europe. The University's student population of 74,000 includes 30,000 international students, of whom more than 17,000 are taught offshore (almost 6,000 at RMIT Vietnam).

We welcome delegates from a number of different countries, including Australia, New Zealand, Austria, Canada, China, the Czech Republic, Denmark, Germany, Hong Kong, Japan, Luxembourg, Malaysia, South Korea, Sweden, the United Arab Emirates, the United Kingdom, and the United States of America.

We hope you will enjoy ACSW2012, and also to experience the city of Melbourne.

Melbourne is amongst the world's most liveable cities for its safe and multicultural environment as well as well-developed infrastructure. Melbourne's skyline is a mix of cutting-edge designs and heritage architecture. The city is famous for its restaurants, fashion boutiques, café-filled laneways, bars, art galleries, and parks.

RMIT's city campus, the venue of ACSW2012, is right in the heart of the Melbourne CBD, and can be easily accessed by train or tram.

ACSW2012 consists of the following conferences:

- Australasian Computer Science Conference (ACSC) (Chaired by Mark Reynolds and Bruce Thomas)
- Australasian Database Conference (ADC) (Chaired by Rui Zhang and Yanchun Zhang)
- Australasian Computer Education Conference (ACE) (Chaired by Michael de Raadt and Angela Carbone)
- Australasian Information Security Conference (AISC) (Chaired by Josef Pieprzyk and Clark Thorburn)
- Australasian User Interface Conference (AUIC) (Chaired by Haifeng Shen and Ross Smith)
- Computing: Australasian Theory Symposium (CATS) (Chaired by Julián Mestre)
- Australasian Symposium on Parallel and Distributed Computing (AusPDC) (Chaired by Jinjun Chen and Rajiv Ranjan)
- Australasian Workshop on Health Informatics and Knowledge Management (HIKM) (Chaired by Keryn Butler-Henderson and Kathleen Gray)
- Asia-Pacific Conference on Conceptual Modelling (APCCM) (Chaired by Aditya Ghose and Flavio Ferrarotti)
- Australasian Computing Doctoral Consortium (ACDC) (Chaired by Falk Scholer and Helen Ashman)

ACSW is an event that requires a great deal of co-operation from a number of people, and this year has been no exception. We thank all who have worked for the success of ACSE 2012, including the Organising Committee, the Conference Chairs and Programme Committees, the RMIT School of Computer Science and IT, the RMIT Events Office, our sponsors, our keynote and invited speakers, and the attendees.

Special thanks go to Alex Potanin, the CORE Conference Coordinator, for his extensive expertise, knowledge and encouragement, and to organisers of previous ACSW meetings, who have provided us with a great deal of information and advice. We hope that ACSW2012 will be as successful as its predecessors.

Assoc. Prof. James Harland

School of Computer Science and Information Technology, RMIT University

ACSW2012 Chair

January, 2012

CORE - Computing Research & Education

CORE welcomes all delegates to ACSW2012 in Melbourne. 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 this week with seven other events - ACE, AISC, AUIC, AusPDC, HIKM, ACDC, and APCCM, which build on the diversity of the Australasian computing community.

In 2012, we have again chosen to feature a small number of keynote speakers from across the discipline: Michael Kölling (ACE), Timo Ropinski (ACSC), and Manish Parashar (AusPDC). I thank them for their contributions to ACSW2012. I also thank invited speakers in some of the individual conferences, and the two CORE award winners Warwish Irwin (CORE Teaching Award) and Daniel Frampton (CORE PhD Award). The efforts of the conference chairs and their program committees have led to strong programs in all the conferences, thanks very much for all your efforts. Thanks are particularly due to James Harland and his colleagues for organising what promises to be a strong event.

The past year has been very turbulent for our disciplines. We tried to convince the ARC that refereed conference publications should be included in ERA2012 in evaluations – it was partially successful. We ran a small pilot which demonstrated that conference citations behave similarly to but not exactly the same as journal citations - so the latter can not be scaled to estimate the former. So they moved all of Field of Research Code 08 “Information and Computing Sciences” to peer review for ERA2012. The effect of this will be that most Universities will be evaluated at least at the two digit 08 level, as refereed conference papers count towards the 50 threshold for evaluation. CORE’s position is to return 08 to a citation measured discipline as soon as possible.

ACSW will feature a joint CORE and ACDICT discussion on Research Challenges in ICT, which I hope will identify a national research agenda as well as priority application areas to which our disciplines can contribute, and perhaps opportunity to find international multi-disciplinary successes which could work in our region.

Beyond research issues, in 2012 CORE will also need to focus on education issues, including in Schools. The likelihood that the future will have less computers is small, yet where are the numbers of students we need?

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 2011; in particular, I thank Alex Potanin, Alan Fekete, Aditya Ghose, Justin Zobel, and those of you who contribute to the discussions on the CORE mailing lists. There are three main lists: csprofs, cshods and members. You are all eligible for the members list if your department is a member. Please do sign up via <http://lists.core.edu.au/mailman/listinfo> - we try to keep the volume low but relevance high in the mailing lists.

Tom Gedeon

President, CORE
January, 2012

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.

2013. Volume 35. Host and Venue - University of South Australia, Adelaide, SA.

2012. Volume 34. Host and Venue - RMIT University, Melbourne, VIC.

2011. Volume 33. Host and Venue - Curtin University of Technology, Perth, WA.

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

ACDC	Australasian Computing Doctoral Consortium
ACE	Australasian Computer Education Conference
ACSC	Australasian Computer Science Conference
ACSW	Australasian Computer Science Week
ADC	Australasian Database Conference
AISC	Australasian Information Security Conference
AUIC	Australasian User Interface Conference
APCCM	Asia-Pacific Conference on Conceptual Modelling
AusPDC	Australasian Symposium on Parallel and Distributed Computing (replaces AusGrid)
CATS	Computing: Australasian Theory Symposium
HIKM	Australasian Workshop on Health Informatics and Knowledge Management

Note that various name changes have occurred, which have been indicated in the Conference Acronyms sections in respective CRPIT volumes.

ACSW and ACE 2012 Sponsors

We wish to thank the following sponsors for their contribution towards this conference.



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



Association for
Computing Machinery

Advancing Computing as a Science & Profession

Association for Computing Machinery,
www.acm.org



RMIT University,
www.rmit.edu.au/



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



**AUSTRALIAN
COMPUTER
SOCIETY**

Australian Computer Society,
www.acs.org.au



Promoting excellence in higher education

Australian Learning and Teaching Council,
www.altc.edu.au

KEYNOTE

The Future of Educational Programming Tools – What Will Come (Or At Least Should Come)

Michael Kölling

School of Computing
University of Kent

`mik@kent.ac.uk`

Abstract

This talk contains no facts. It is made up entirely of my speculations and opinions about what will (or should) happen in the near future of our discipline: Computer Science Education. Since my own personal background is in the area of educational software tools, much of it will be commentary on and speculation about the future of software tools. However, I will not let my potential ignorance of other topics stop me from making comments on the wider discipline.

Since I am not any more psychic than the average person in the audience, I might be completely wrong with any predictions, and this talk might come down to no more than a collection of unprovable opinions. However, even if people disagree with most of what I say, I hope that many get at least some enjoyment out of their disagreement.

CONTRIBUTED PAPERS

Perceptions of a gender-inclusive curriculum amongst Australian Information and Communications Technology academics

Tony Koppi Madeleine Roberts Golshah Naghdy

Faculty of Informatics, University of Wollongong, Australia

tkoppi@uow.edu.au

mrhr01@uowmail.edu.au

golshah@uow.edu.au

Abstract

The lack of female enrolments in ICT is widely recognised and has prompted a range of strategies to attract more women, most of which do not include curriculum changes at any level. Research suggests that there are aspects of the ICT curriculum that could appeal to females, particularly in relation to benefits to society and humanity in general, and that including these considerations in the curriculum would be of interest to all students. The perceptions of a gender-inclusive ICT curriculum in Australia have been ascertained from a survey and forum discussions of ICT academic managers and leaders of ICT learning and teaching. Although a significant proportion of the surveyed academics recognises that different features of the ICT curriculum appeal to males (mainly technology) and females (mainly the benefits of the technology to humanity) this has not translated into the practical implementation of a gender-inclusive curriculum in most institutions. Most respondents would welcome informative guidelines on developing a gender inclusive curriculum.

Keywords: Gender inclusive curriculum, perceptions, ICT, academics

1 Introduction

This paper is concerned with the perceptions of a gender-inclusive ICT curriculum held by Australian ICT academic staff as derived from survey and forum data. Gender is an issue in ICT – as it is in the related Engineering discipline (Mills et al. 2010) – because ICT has a male-dominated culture (Vilé and Ellen, 2008). Australian Women in Information Technology (OzWIT 2006) reported that 15% of ICT workers were female and that the trend in the employment of female ICT workers is downwards, with similar numbers and trends in Europe (Valenduc and Vendramin 2005). Lewis et al. (2006) reported that the proportion of women in many ICT courses in Australia is less than 15%. Some research

suggests that few women enrol in ICT because of the perceived masculine stereotype (Cory et al. 2006) which is reinforced by stereotypical high school teacher behaviour (Dee 2007), and compounded by the differing societal attitudes and influences brought to bear on boys and girls during their development (Dingel 2006). Evidence shows that (perhaps as a result of these influences) anxiety and lack of confidence in using computers is more prevalent among women than men (Volman and van Eck 2001), even amongst experienced users (Beyer et al. 2003, Broos 2005).

Part of the definition of a gender inclusive ICT curriculum is one that is inclusive of social and human concerns and portrays technology in that context without lessening the content (Koppi et al. 2010). It is suggested that current ICT curricula that are focussed on technology-centred topics are biased towards male students (Lewis et al. 2007, Lewis et al. 2006, Miliszewska and Moore 2010). Other aspects of a gender inclusive curriculum include: respecting every student as an individual and enabling them to reach their potential; recognising and accommodating differences in interests, experiences and circumstances of all students; and adjusting the curriculum in response to feedback (Mills et al. 2010). Furthermore, it has been argued that gender inclusivity in decision making in the ICT context may result in more balanced and favourable outcomes (Cukier et al. 2002).

The apparent bias towards a masculine-oriented curriculum in a male-dominated culture may be a contributing factor to the attrition of females from higher education ICT courses. There is general concern about high attrition rates from ICT (Connolly and Murphy 2005, Koppi and Naghdy 2009, McMillan 2005), including that of women (Miliszewska et al. 2006, Sheard et al. 2008). The application of a gender-inclusive curriculum is pertinent to this situation. This paper examines the prevailing perception of a gender inclusive curriculum in ICT in the Australian higher education context.

The notion of ‘Curriculum’ is a broad concept (Hicks 2007), as are the influences on the curriculum when situated within a male-dominated ICT culture that is decades old and extends from primary school to the workplace and western society at large. Hicks (2007) points out that the ‘Curriculum’ is part of this broad temporal and spatially connected structure, no part of which exists in isolation. To adopt such a holistic context

Copyright © 2012, Australian Computer Society, Inc. This paper appeared at the 14th Australasian Computing Education Conference (ACE 2012), Melbourne, Australia, January–February 2012. Conferences in Research and Practice in Information Technology, Vol. 123. M. de Raadt and A. Carbone, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

of the ICT higher education curriculum, this paper reports on what universities are doing with respect to attracting (reaching into schools) and retaining students, as well as the prevailing perceptions and practices of an inclusive curriculum (or otherwise) that imprints on postgraduate professionals in education, industry and government and feeds back to students and parents.

A survey and a related workshop were carried out as part of a recent project (Ogunbona 2009) funded by the Australian Learning and Teaching Council (ALTC) and comprised a team from four Australian universities: Wollongong (lead institution), Murdoch, Swinburne and Queensland.

2 Survey and Analysis Methods

2.1 Survey Groups

The aim of the survey developed by the project team was to obtain an understanding of the representative view held by Australian ICT academic staff about a gender-inclusive curriculum. The first targeted group was the heads of ICT units at all Australian universities who were members of the Australian Council of Deans of ICT. A series of four approaches was used in order to obtain their participation in the data-gathering exercise: a paper-based survey was mailed to each university representative on the Council; a fortnight later an email reminder with the survey attached was sent to the same people; a telephone follow up was undertaken two weeks later and finally they were sent an invitation to complete the survey online. As a result of these efforts a total of 22 completed surveys were received from 18 universities (a few ICT heads had distributed the survey to other ICT heads internally). The second group to complete the same survey was the Associate Deans for Learning and Teaching (or their equivalent) in ICT at an Australian Council of Deans of Information and Communications Technology (ACDICT) forum of 35 attendees representing 25 universities, and 24 completed surveys were received. In addition, a workshop session on gender issues was held at the forum and the recorded discussions were also used to inform the project. The total of 46 completed surveys and forum deliberations are considered as representative of Australian ICT academia concerning a gender-inclusive ICT curriculum.

2.2 Survey Analysis

The survey consisted of a series of questions to be rated on a 5-point Likert Scale, where a tick was sufficient to indicate the response, to provide quantitative data. In addition, a number of open-ended questions were presented to allow free text responses. These free text entries were read several times to enable the coding and categorisation of responses which were then counted to enable quantitative comparisons. This qualitative data analysis method was informed by the work of Boyatzis (1998), and Bogdan and Bicklen (2002).

2.3 Workshop Discussions

During the ACDICT forum workshop, participants were organised into small groups to discuss gender issues in ICT and invited to focus particularly on the possible reasons for the lack of women in ICT, and on the nature

of a gender-inclusive ICT curriculum. Their deliberations were summarised on paper by each group, collected and later compiled by the facilitator. Plenary discussions were summarised and typed for the whole group to see on screen and edit at the time.

3 Findings

3.1 Enrolment Trends of Women in ICT

About half of the survey respondents noted that undergraduate enrolments of women are steady (with some respondents commenting that these are small numbers) and about one third noted that the numbers of women were falling. Only five respondents indicated that the number of female undergraduate enrolments was increasing at their university. Most respondents (76%) noted that they were trying to increase the enrolments of women in ICT but few (18%) noted that their strategies to do so were effective.

3.2 Strategies for Increasing the Enrolment of Females in ICT

3.2.1 Current Strategies from Survey Responses

The survey included the open-ended statement: "Our strategy for increasing the enrolment of women in ICT is:". The responses indicated that most of the strategies being used were apparently not effectively contributing to increased enrolments of females. Presumably, the situation could be worse without any strategies aimed at encouraging females to enrol in ICT. This presumption may explain why similar strategies, at a range of universities, have been used repeatedly over the years and why it is that Craig (2010) has recently recognised that strategies intended to attract females must be more formally structured and stringently assessed.

The survey data revealed that the main practices in targeting female students from years 9–12 are threefold: (1) female ICT staff or students visiting schools as role models or ambassadors; (2) inviting female students to participate in various ICT activities at universities, such as engineering or programming workshops; and (3) female specific events, e.g., Go Girl Go For IT, supported by Victorian ICT for Women (2010) held bi-annually at Deakin University since 2006. This event also includes invitations to careers advisors and teachers, has been attended by over 2000 people and is, apparently, a successful activity (now called 'Digital Divas') in improving female perceptions of a career in ICT (Lang et al. 2010).

Other strategies mentioned by respondents include scholarships for females enrolling in ICT. Notwithstanding the success of marketing events or inducements, survey respondents noted that females in high school still have negative experiences with ICT which probably significantly contribute to the overall low enrolment rate of females in ICT higher education. Not one respondent mentioned whether or not the high school curriculum is gender inclusive and the impact this may have on female experiences. It would seem that for a subject with such a strong gender imbalance, trying to attract more females is akin to treating the symptoms and not the cause, which still remains to be clearly articulated.

While most of the strategies being used are apparently not contributing to increased enrolments of females, the situation would presumably be worse without any strategies aimed at encouraging females to enrol in ICT. This rationale may explain why similar strategies, at a range of universities, have been used without modification over the years.

3.2.2 Further Desirable Strategies from Survey Responses

A follow up open-ended statement was “Additional activities that we should be doing to attract more women into ICT are:”. Common responses included: working further with females in high schools; emphasizing employment opportunities; and improving perceptions of the ICT profession generally. A few respondents mentioned working with high school teachers and ensuring that relevant enabling subjects are taught. One person mentioned an inclusive curriculum in terms of being inclusive of different interest areas, one person noted gender inclusive projects, and one respondent noted that social and business dimensions should be emphasized. No one mentioned the desirability of a gender inclusive curriculum *per se*.

Apart from university strategies, when asked what else could be done to entice more women into ICT, it was observed that on-going attempts by the ACS (Australian Computer Society) professional body were largely ineffective, and that the Federal Government should be more forthcoming with financial incentives to encourage more students into ICT and in helping to drive cultural change through government policies. High schools were still seen as the primary focus for changing the perceptions of ICT and that women professionals in industry should be more involved in school visits, as well as industry engaging in more marketing and promotional activities in general. Revising the high school ICT curriculum was seen as essential, and suggestions included: less focus on technology; promoting the value of mathematics; emphasising the communication and ‘soft skills’ aspects of ICT; and having gender-inclusive projects.

3.2.3 Workshop Deliberations

During the workshop event, participants discussed the issue of the small number of women attracted to ICT. It was observed that there was a greater proportion of female students in international cohorts than domestic students and that this reflected cultural differences. The Australian ICT culture was described as male-centred (e.g., advertisements portraying men in the profession; lack of female role models; and a perception of being unsuitable as a female career), geeky, and technology-centred rather than outcome focused. High schools were thought to be reflecting this culture, providing a narrow curriculum that focused on technology tools and lacking creativity, diversity and failing to present the broad ranging functions and roles of ICT. Furthermore, high school teachers and careers advisors were thought to have a limited understanding of ICT and its potential. Apart from gender issues, the high school ICT curriculum was considered to lack inclusivity in terms of content, scope

and application, and probably contributed to domestic students deciding, early in their high school education (before year 10), against a career in ICT.

The survey results (essentially of individuals) concerned with attracting women into ICT revealed that a minority of respondents demonstrated awareness of notions of a gender-inclusive curriculum. By contrast, the group discussions (which had women at almost every table) at the workshop event concerned with the lack of women in ICT revealed a greater awareness of gender inclusivity. This may have been due to the specific gender theme and individual concerns expressed through the group work.

3.3 The Gender-Inclusive ICT Curriculum: Theory and Practice

The survey data (Table 1) revealed that 24% of respondents agreed with the statement that there is a link between having a gender-inclusive curriculum and the low proportion of women studying ICT, while 41% disagreed. 28% agreed that they make an effort to have an explicitly gender-inclusive curriculum, and 24% agreed that an ICT curriculum that appeals to women would be different to one that appeals to men. The majority of respondents (89%) agreed that they would welcome informed guidelines on the practical implementation of a gender-inclusive ICT curriculum. In addition, 62% of respondents agreed with the statement that they are unsure of what a gender-inclusive ICT curriculum would really look like – indicating that the majority of ICT academic staff is unclear about the nature of a gender-inclusive curriculum.

Statements regarding an ICT curriculum	SD	D	N	A	SA
We are unsure of what a gender-inclusive ICT curriculum would really look like	2	9	6	21	7
An ICT curriculum that appeals to women would be different to one that appeals to men	6	15	13	10	1
We make an effort to have an ICT curriculum that is explicitly gender-inclusive	1	20	10	8	4
There is a link between having a gender-inclusive curriculum and the low proportion of women studying ICT	3	14	14	10	0
We would welcome informed guidelines on the practical implementation of a gender-inclusive ICT curriculum	1	0	4	25	15

Table 1: Compiled responses to survey statements about a gender inclusive curriculum (SD = Strongly Disagree to SA = Strongly Agree)

Margolis and Fisher (2002) have noted that, on the whole, women have a different perspective of computer science (a part of ICT) to males, and the awareness of any such difference amongst Australian ICT academics was explored in the survey by asking about the features of the ICT curriculum that would appeal to females and males.

3.3.1 Perceptions of an ICT Curriculum that would Appeal to Females

Respondents to the survey were asked about the features of an ICT curriculum that appeal to females (Table 2) and 35% of respondents either left this field blank or stated that they didn't know. Only 11% stated that there was either no difference in what appealed to males and females, or that generalising was inadvisable. A little over half of the respondents gave some indication of what appealed to females in the ICT curriculum. Of the total, 30% of respondents noted that it was the 'people' side of ICT that appealed to women, using words such as: 'people', 'social', 'community', 'collaborative', 'society' and 'humanity'. These responses are consistent with the conclusions of Courtney, Timms and Anderson (2006) and Craig, Fisher and Lang (2007) that females are particularly interested in the people part of the profession. A total of 11% of the respondents mentioned that a focus on communication (interpersonal rather than technology) appeals to females. That there are differences in communication between males and females have been reported many times (Monaghan and Goodman 2006, Still 2006, Wood 2005), and one would therefore expect it to be a relevant factor. A total of 11% of respondents also noted that creativity and problem solving, especially in a global or big-picture context, would appeal to females.

Category of responses regarding curriculum aspects appealing to females	Number (n = 46)	%
Blank	12	26
Don't know	4	9
No difference	2	4
Unsafe/unwilling to generalise	3	7
Soft/softer skills	2	4
People/social/community/society/humanity/collaborative	14	30
Communication	5	11
Creative/problem-solving	5	11
Technology	1	2

Table 2: Categories of responses to the survey question of features of the ICT curriculum that appeal to females

The implication is that these responses are also related to the people interests that appeal to women. With respect to the skills of problem solving and creativity *per se*, there is apparently little if any difference between males and females and that context has a strong influence on the expression of many skills (Hyde 2005). It is interesting to note that only one respondent mentioned technology itself as appealing to women, and there was no mention of laboratory work.

3.3.2 Perceptions of an ICT Curriculum that would Appeal to Males

From the survey, Table 3 shows the categories and responses to the open-ended question about the features of an ICT curriculum that appeal to males.

35% of respondents either left this field blank or stated that they didn't know. Of the total responses, 50%

mentioned some aspect of technology or playing with technology with the words: 'hardware', 'networking', 'programming', 'games', 'competitions', 'technology', and 'shooting'. Only two people thought it was difficult to generalise and one noted that not all men like playing with technology.

Various other aspects of the curriculum that appealed to males were mentioned, such as laboratory work, solo efforts, creativity, problem solving, design, building, and project management. There was no mention of males having an interest in people or the application of technology to social issues. This view that males in ICT tend to be more interested in the technology than social and human concerns has been reported elsewhere (Lewis et al. 2007, Lewis et al. 2006, Moore et al. 2005).

Category of responses regarding curriculum aspects appealing to males	Number (n = 46)	%
Blank	13	28
Don't know	3	7
Difficult/unsafe to generalise	3	7
Hardware, networking, programming, games, competitions, technology, shooting, play,	23	50

Table 3: Categories of responses to the survey question of features of the ICT curriculum that appeal to males

3.3.3 Workshop discussions on a Gender-Inclusive ICT curriculum

Three groups of six discussed the issues concerning a gender inclusive curriculum before the plenary session (35 attendees). Two of the three groups contained both men and women and their group summaries identified the perception that females were concerned with human issues and that they needed to see the benefits of ICT to the community on a broad range of fronts such as health, education and the environment. The all-male group was unable to speculate on what a gender-inclusive curriculum would look like.

The plenary session reinforced the perception that women need to know 'why' and have people and community concerns about how ICT can solve people's problems: women have human concerns about how technology can build a better world. These greater humanity concerns of females expressed at the workshop are consistent with the literature cited above. It was noted that business ICT degrees have a greater proportion of women than technology-focused degrees. It was also concluded that the ICT curriculum problem starts in high school.

3.3.4 Measures to Ensure a Gender-Inclusive Curriculum

From the survey, Table 4 shows the categories and responses to the open-ended question about the measures taken to ensure a gender-inclusive ICT curriculum.

39% of respondents either left this blank or indicated that they didn't know or were unsure. 20% noted that they had done nothing to make their curriculum gender-inclusive or that it was not gender-inclusive. A few noted that 'soft skills' such as teamwork and communication

had been increased, and that stereotypes and male-centred examples were avoided. A few also noted that technology was presented as part of a systems or society or people perspective. Allowing students to select their own projects of interest was seen as part of being gender-inclusive. One person also mentioned the use of female role models.

Measures taken to ensure a gender-inclusive ICT curriculum	Number (n = 46)	%
Blank	14	30
Don't know/unsure	4	9
Done nothing or non-existent	9	20
Increased soft skills such as teamwork, communication	3	7
Avoiding stereotypes and male-centred examples	3	7
Technology as part of the system/society/people perspective	3	7
Project choice	2	4

Table 4: Categories of responses to the survey question of the measures taken to ensure a gender-inclusive ICT curriculum

4 Discussion

4.1 Enrolment Strategies in Relation to a Gender-Inclusive Curriculum

In Australia, it is widely recognised that the ICT culture from high schools through to industry is male-dominated and that the proportion of females studying ICT is small (Craig et al. 2007, Lang et al. 2010, Lasen 2010, Lewis 2006, McLachlan 2010, Miliszewska 2010, Young, 2003). Results from this study have shown that the higher education ICT curriculum is largely reflective of that culture and that a gender-inclusive curriculum is not well understood or established. It would seem that the culture and the curriculum are related, mutually reinforcing and perpetuating. However, only 24% of survey respondents agreed with the statement that there is a link between having a gender-inclusive curriculum and the low proportion of women studying ICT. It is therefore not surprising that strategies to increase female participation in ICT higher education over the years have not changed much and have largely been unsuccessful.

The survey revealed a range of intervention strategies employed by universities in an attempt to encourage more females into ICT, including using female ambassadors and female-only events at universities. A likely reason for the lack of success of these intervention strategies is that their evaluation is apparently not usually carried out (Craig et al. 2011).

Other suggested strategies included more effective ACS activities (unspecified) or greater Federal Government financial incentives and policies to bring about cultural change. However, if the curriculum remains male-centred, the culture is unlikely to change and enrolment strategies will continue to be largely ineffective fringe activities with regard to increasing the proportion of female enrolments. They may, in fact, be effective in maintaining the small numbers.

4.2 Gender-Inclusive ICT Curriculum Perceptions

The majority of survey respondents agreed that they were unsure about what a gender-inclusive ICT curriculum would really look like, yet a relatively large proportion identified the features of an ICT curriculum that appeal to males and females that are consistent with other published findings. About 40% of survey respondents noted that females tend to be more interested in the people side of the discipline and the skills required to benefit society and humanity at large, a view that is supported by the literature (e.g. Bissell et al. 2002, Margolis and Fisher 2002, Tillberg and Cohoon 2005, Courtney et al. 2006, Craig et al. 2007). About 50% of survey respondents also noted that males tend to be more interested in the technology rather than human concerns, which is a view also supported by Moore, Griffiths and Richardson (2005); Lewis et al. (2006); and Lewis et al. (2007). Workshop attendees also expressed similar views. These findings are not meant to imply that different perspectives are as a result of exclusively masculine or feminine characteristics, rather that there is a tendency broadly related to gender resulting from societal influences which dictate what is feminine and what is masculine (Dingel 2006, Jaworski and Coupland 1999, Seymour and Hewitt 1997). Undoubtedly there are women interested in the technology *per se* and men interested in the social application.

If the curriculum creators are also affected by these societal influences which dictate gender, and are unaware of them to an extent (as suggested from these survey results) then gender stereotypes are being reinforced and contributing to the lack of a gender inclusivity in the teaching of ICT.

Almost half of the surveyed ICT academic staff is aware of gender differences and interests in the discipline yet a much smaller proportion indicated that any practical measures addressing these issues were in place. There appears to be a considerable gap between what is known (or at least suspected) and practiced. This is supported by the fact that 89% of survey respondents expressed the desire for informed guidelines on the practical implementation of a gender-inclusive ICT curriculum.

The issue of a gender inclusive curriculum is not confined to ICT. Engineering (a related discipline) is also beginning to address these issues (Mills et al. 2010). While Engineering and ICT gender curriculum issues may be perceived as being different, the similarities probably outweigh the differences. A significant factor in making the curriculum more gender-inclusive is concerned with emphasising the *context* of the technology so that all students may readily perceive its relevance to improving society (Koppi et al. 2010). Other curriculum aspects such as student experiences, forms of assessment, learning and teaching methods and the learning environment are also part of gender inclusive considerations (Mills et al. 2010). The deliberate implementation of gender inclusive practices has been shown to make significant differences in attracting and benefiting all students (Margolis and Fisher 2002). However, changing the curriculum depends on many factors, such as individuals, politics and fashion; and

academic merit and curricula practiced elsewhere are not necessarily major concerns (Gruba et al. 2004).

5 Conclusion

While there is a broad appreciation amongst a significant proportion of ICT academics in Australia that there are different gender perspectives and interests in ICT, this perception has not necessarily translated into a gender-inclusive curriculum. The desire for such a curriculum has been expressed even though the practical development and implementation is unclear and there are different perceptions of what a gender-inclusive curriculum would entail.

Given the gender balance amongst ICT academics, it is likely that the survey results reflect a male perspective.

Research has shown that a comprehensive approach to curriculum design needs to be adopted to make it more inclusive. This is likely to be a protracted process because of innate conservatism and the slow pace of curriculum change in the sector (Gruba et al. 2004).

Most universities make a special effort to attract more female students into ICT, even though the prevailing culture is male-dominated, the majority of academic staff is male and the curriculum is apparently largely biased towards programmes more appealing to males. The culture is reinforced by the lack of a gender-inclusive curriculum which can only perpetuate the imbalance. Significant increased enrolments of females cannot be expected to occur if bias persists at all levels. What is needed is a new curriculum that will produce a new style of ICT professional so that the cycle can evolve.

6 Acknowledgements

The following project team members contributed to the survey design: Jocelyn Armarego (Murdoch), Paul Bailes (UQ), Tanya McGill (Murdoch), Fazel Naghdy (UOW), Philip Ogunbona (UOW) and Chris Pilgrim (Swinburne).

7 References

- Australian Women in Information Technology (OzWIT) (2006): Statistics for the IT Industry in 2006. http://www.ozwit.org/version_three/index.php?option=com_content&task=view&id=15&Itemid=34. Accessed 6 April 2011.
- Beyer, S., Rynes, K., Perrault, J., Hay, K. and Haller, S. (2003): Gender differences in computer science students. *SIGCSE Bulletin* **35**(1):49-53.
- Bissell, C., Chapman, D., Herman, C. and Robinson, L. (2002): Some gender issues in the teaching of the information and communication technologies at the UK Open University. <http://technology.open.ac.uk/tel/people/bissell/gender.pdf>. Accessed 23 May 2009.
- Bogdan, R.C. and Biklen, S.K. (2002): *Qualitative research for education: an introduction to theories and methods*. 4th edn. London, Allyn & Bacon.
- Boyatzis, R.E. (1998): *Transforming qualitative information: thematic analysis and code development*. Thousand Oaks California, Sage Publications.
- Broos, A. (2005): Gender and information and communication technologies (ICT) anxiety: male self-assurance and female hesitation. *CyberPsychology & Behavior* **8**(1):21-31.
- Connolly, C. and Murphy, E. (2005): Retention initiatives for ICT based courses. *Proc. Frontiers in Education*, Indianapolis, Indiana, S2C-10.
- Cory, S.N., Parzinger, M.J. and Reeves, T.E. (2006): Are high school students avoiding the information technology profession because of the masculine stereotype? *Information Systems Education Journal* **4**(29):3-13.
- Courtney, L., Timms, C. and Anderson, N. (2006): "I would rather spend time with a person than a machine": qualitative findings from the girls and ICT survey. In *Quality and Impact of Qualitative Research*. 51-57. Ruth, A. (ed). Griffith University, Australia.
- Craig, A. (2010): *Attracting women to computing: a framework for evaluating intervention programmes*. Saarbrücken, Germany, VDM Publishing.
- Craig, A., Fisher, J. and Lang, C. (2007): ICT and girls: the need for a large-scale intervention. *Proc. 18th Australasian Conference on Information Systems*, Toowoomba, Australia, 761-769.
- Craig, A., Fisher, J., Forgasz, H. and Lang, C. (2011): Evaluation Framework Underpinning the Digital Divas Programme. Paper presented at the Innovation and Technology in Computer Science Education Conference (ITiCSE'11, June 27-29), Darmstadt, Germany, 313-317.
- Cukier, W., Shortt, D. and Devine, I. (2002): Gender and Information Technology: Implications of Definitions. *SIGCSE Bulletin*, 34(4), 142-148.
- Dee, T.S. (2007): Teachers and the gender gaps in student achievement. *Journal of Human Resources* **XLII**(3):528-554.
- Dingel, M.J. (2006): Gendered experiences in the science classroom. In *Removing Barriers: Women in Academic Science, Technology, Engineering and Mathematics*. 161-176. Bystydzienski, J.M. and Bird, S.R. (eds). Bloomington, Indiana University Press.
- Gruba, P., Moffat, A., Sondergaard, H. and Zobel, J. (2004): What Drives Curriculum Change?. In *Proc. Sixth Australasian Computing Education Conference (ACE2004)*, Dunedin, New Zealand. CRPIT, 30. Lister, R. and Young, A.L., Eds., ACS. 109-117.
- Hicks, O. (2007): Curriculum in higher education in Australia - hello? <http://altcexchange.edu.au/system/files/Curriculum%20in%20Higher%20Education%20-%20HERDSA%20Full%20Paper.doc>. Accessed 12 Aug 2010.
- Hyde, J.S. (2005): The gender similarities hypothesis. *American Psychologist* **60**(6):581-592.
- Jaworski, A. and Coupland, N. (eds) (1999): *The discourse reader*. 2nd edn. London, Routledge.
- Koppi, T. and Naghdy, F. (2009): Managing educational change in the ICT discipline at the tertiary education level. <http://www.altc.edu.au/system/files/resources/DS6-600%20Managing%20educational%20change>

- %20in%20the%20ICT%20discipline%20March%202009.pdf. Accessed 20 March 2010.
- Koppi, T., Sheard, J., Naghdy, F., Edwards, S.L. and Brookes, W. (2010): Towards a gender inclusive ICT curriculum: a perspective from graduates in the workforce. *Computer Science Education* **20**(1):1-18.
- Lang, C., Craig, A., Fisher, J. and Forgasz, H. (2010): Creating digital divas: scaffolding perception change through secondary school and university alliances. *Proc. 15th Annual Conference on Innovation and Technology in Computer Science Education*, Ankara, Turkey.
- Lasen, M. (2010): Education and career pathways in information communication technology: what are schoolgirls saying? *Computers and Education* **54**(4):1117-1126
- Lewis, S., Lang, C. and McKay, J. (2007): An inconvenient truth: the invisibility of women in ICT. *Australasian Journal of Information Systems* **15**(1):59-76.
- Lewis, S., McKay, J. and Lang, C. (2006): The next wave of gender projects in IT curriculum and teaching at universities. *Proc. Eighth Australasian Computing Education Conference*, Hobart, Tasmania, **52**.
- Margolis, J. and Fisher, A. (2002): *Unlocking the clubhouse: women in computing*. Cambridge, Massachusetts, MIT Press.
- McLachlan, C., Craig, A. and Coldwell, J. (2010): Student perceptions of ICT: a gendered analysis. *Proc. 12th Australasian Computing Education Conference*, Brisbane, Australia.
- McMillan, J. (2005): *Course change and attrition from higher education*. Canberra, Department of Education Science and Training.
- Miliszewska, I. and Moore, A. (2010): Encouraging girls to consider a career in ICT: a review of strategies. *Journal of Information Technology Education* **9**:143-166.
- Miliszewska, I., Barker, G., Henderson, F. and Sztendur, E. (2006): The issue of gender equity in computer science – what students say. *Journal of Information Technology Education* **5**:107-120.
- Mills, J., Ayre, M. and Gill, J. (2010): *Gender inclusive engineering education*. New York, Routledge.
- Monaghan, L. and Goodman, J. (eds) (2006): *A cultural approach to interpersonal communication*. Oxford, UK, Blackwell Publishing Ltd.
- Moore, K., Griffiths, M. and Richardson, H. (2005): Moving in, moving up, moving out? a survey of women in ICT. *Proc. Third European Symposium on Gender and ICT: Working for Change*, Manchester, UK.
- Ogunbona, P. (2009): ALTC Project PP9-1274, Addressing ICT curriculum recommendations from surveys of academics, workplace graduates and employers. <http://www.altc.edu.au/project-addressing-ict-curriculum-recommendations-uow-2009>. Accessed 6 June 2010.
- Seymour, E. and Hewitt, N.M. (1997): *Talking about leaving: why undergraduates leave the sciences*, Boulder, Colorado, Westview Press.
- Sheard, J., Carbone, A., Markham, S., Hurst, A.J., Casey, D. and Avram, C. (2008): Performance and progression of first year ICT students. *Proc. Tenth Australasian Computing Education Conference*, Wollongong, Australia.
- Still, L.V. (2006): Gender, leadership and communication. In *Gender and Communication at Work*. 183-194. Barrett, M. and Davidson, M. Aldershot, UK, Ashgate Publishing Ltd.
- Tillberg, H.K. and Cohoon, J.M. (2005): Attracting women to the CS major. *Frontiers* **26**(1):126-140.
- Valenduc, G. and Vendramin, P. (2005): Work organisation and skills in ICT professions: the gender dimension. *Proc. ICT, the Knowledge Society and Changes in Work*, Den Haag, Netherlands.
- Vilé, S. and Ellen, J. (2008): Australian computer society: women members survey 2008: a thematic analysis. <https://www.acs.org.au/acswomen/docs/ACSWomenMembersSurveyThematicAnalysisReportAugust2008.pdf>. Accessed 10 Sept 2010.
- Volman, M. and van Eck, E. (2001): Gender equity and information technology in education: the second decade. *Review of Educational Research* **71**(4):613-634.
- Victorian ICT for Women (2010): Go Girl Go For IT. <http://www.vicictforwomen.com.au/www/html/334-go-girl-go-for-it-2010-.asp>. Accessed 17 Sept 2010.
- Wood, J.T. (2005): *Gendered lives: communication, gender, and culture*. Belmont, California, Thomson Wadsworth.
- Young, J. (2003): The extent to which information communication technology careers fulfil the career ideals of girls. *Australasian Journal of Information Systems* **10**(2):115-125.

Attrition from Australian ICT Degrees – Why Women Leave

Madeleine R. H. Roberts

School of Information Systems and Technology
University of Wollongong, NSW

mrhr01@uowmail.edu.au

Tanya J. McGill

School of Information Technology
Murdoch University
Murdoch, WA

t.mcgill@murdoch.edu.au

Peter N. Hyland

School of Information Systems and Technology
University of Wollongong, NSW

phyland@uow.edu.au

Abstract

Student attrition is of particular concern in the field of ICT because the industry faces staffing shortfalls, generally and a noticeable lack of female employees. This paper explores the reasons female students give for leaving their ICT courses. An online survey of early leavers from four Australian universities was conducted. The results show that, for many female students, it is a combination of issues that leads to their withdrawal. Contrary to expectations, few female ex-students had experienced serious life events that necessitated their withdrawal or indicated that negative behaviour or attitudes had contributed to their decision to leave. More commonly female participants cited reasons associated with their lack of expected background knowledge and with issues related to the course. Recommendations are made to address issues that could be mitigated by university action.

Keywords: Gender; female; ICT education; student attrition; student retention.¹

1 Introduction

Student attrition is of particular concern in the field of ICT because the industry faces staffing shortfalls (ACS 2008, e-skills UK 2011). Women have long been under represented in ICT employment and in ICT courses (Logan & Crump 2007), and there has been little sign of improvement (Gras-Velazquez et al. 2009). For example, only 1,997 female students commenced an ICT bachelor degree in Australia in 2009 compared to 9,106 male students (DEEWR 2011a). Given the low number of females entering ICT courses, it is essential that those that do enroll successfully complete their studies. Figures

from DEEWR show that in Australia approximately 16% of commencing female students leave their ICT course per year (DEEWR 2011b). This paper explores the reasons female students give for leaving their ICT courses and makes recommendations to improve their retention.

Attrition is the central theme of this paper and there are numerous definitions of its meaning from Seidman's simple "diminution in numbers of students resulting from lower student retention" (Seidman 2005, p. 92) to Hinton's (2007) comprehensive identification of nine forms of attrition. In this study the term attrition is used to indicate the loss of students from ICT courses either because: they leave the institution altogether or because they transfer to another non-ICT course at the same institution. It is thus used at both the institutional level and the course level.

ICT courses have very high attrition rates. An Australian study (Marks 2007) identified ICT as having the highest attrition rate with approximately one third of students leaving. A similar UK study (Bailey & Borooah 2007) found a 28% attrition rate. In comparison, medicine had an attrition rate of less than 5%, and education roughly 14%. Attrition rates of female students from ICT courses appear to be consistent with those of males (DEEWR 2011). However given the low number of female students starting, the industry cannot afford to lose them, and universities should do all that they can to retain them.

Numerous studies have investigated the reasons for attrition from tertiary education around the world. Many of these have focused on only one reason at a time, such as financial aid (Stater 2009), the effect of boredom (Mann & Robinson 2009) or students with dependent children (Marandet & Wainwright 2009) while others have attempted to cover a spectrum of reasons. Hovdhaugen (2009), for example, focused on both personal characteristics (gender, age, social background and prior academic achievement) and student goals and motivation once enrolled. The study found that personal characteristics explained withdrawal more effectively than student goals or motivation, while the latter largely

Copyright © 2012, Australian Computer Society, Inc. This paper appeared at the Fourteenth Australasian Computing Education Conference (ACE2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 123. M. De Raadt and A. Carbone, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

explained the reasons for transfer. Bennett (2003) and Bailey and Borooah (2007) also studied the role of personal characteristics in attrition and both studies confirmed the importance of financial hardship. Some authors (Hagedorn, 2005; Kramer, 2007; Nora et al., 2005; Price et al., 1992; Tinto, 1993) have chosen to examine and discuss students as a homogenous entity while others have recognised the differences in experiences for male and female students (Barrow et al., 2009; Charles & Bradley, 2006; Manis et al., 1989; Seymour & Hewitt, 1997) and the need to identify, more specifically, why their experiences are different and what effects that has upon their decision to stay or go. Early models such as those proposed by Tinto (1975) and Bean (1980) have proved useful in understanding attrition, and have been extended by various authors to better predict and understand the phenomenon. Cabrera et al. (1993) investigated whether Tinto's Student Integration Model and Bean's Student Attrition Model could be merged. As well as confirming relationships among the commitment, social and academic integration factors they also found support for the effect of external factors such as encouragement from friends and family on the student's commitment to the institution.

In addition to studies focussing on attrition across a range of disciplines, there have been a number of studies focussing on attrition in ICT degrees. Barker et al. (2009) investigated factors relating to the social experience in computer science by exploring the types of interactions students had with peers, teachers and staff, and found that positive student-student interaction could be enhanced through the use of collaborative learning experiences in the classroom. Prior experience in programming was found to be an important predictor of intention to continue in computer science, however, technical ability appeared to be less important than soft skills (Lewis et al. 2008).

Whilst ICT faces many of the same issues as other disciplines, factors such as the low numbers of female students enrolling, and reports of higher female attrition rates (Barker et al. 2009) differentiate it. There is some evidence that these are linked, as an increase in the proportion of females has been shown to reduce attrition (Cohoon 2001). This finding is unremarkable, however, when the culture of computing is considered. Margolis and Fisher (2002) amply demonstrate the existence of a "Clubhouse" in computing which will, without any intention on the part of the male members, exclude women on the basis of numerical superiority alone. Previous research has shown that, while female ICT students do not appear to differ from male students in terms of their academic ability to understand the material, they lack confidence in their ability to do so (Beyer et al. 2003) and they may also have had less previous ICT experience (Cohoon & Aspray 2006). This view of female "deficiency" (Henwood 2000) must, however, be challenged and questions must be asked about why the computing curriculum contains assumptions about previous knowledge and experience. It is also imperative to investigate the actual reasons for women's loss of confidence in their abilities, since "lack" and "loss" are very different descriptors: the former indicating no confidence while the latter indicates confidence that has

been eroded. Margolis and Fisher (2002) convincingly demonstrate the erosion of confidence experienced by female computer science students attending Carnegie Mellon's prestigious Computer Science School where their very presence is questioned by fellow students who boast of their abilities and achievements, resulting in female students' disillusionment and waning enthusiasm.

Other studies specific to attrition in ICT have explored whether technical skills and emotional intelligence contribute to students' "affinity" with their major (Lewis et al. 2008). The researchers defined technical skills as the ability to: solve problems through abstraction and decomposition; develop algorithms; programme; and test. Emotional intelligence was defined as the ability to: understand emotion; control and express emotion; and use emotion in finding solutions. The study found that females with technical skills and emotional intelligence were most likely to remain in their major and that incorporating more soft skills into the curriculum would not only benefit all students but also create graduates far better suited to the current requirements of industry which include the ability to contribute effectively in teamwork.

The outcomes of these many studies suggest that attrition is influenced by both the personal characteristics of students and the educational environment. Some factors apply across many disciplines, and some are more discipline specific. Some factors appear to be gender specific. While some factors, such as a student's personal life and financial pressures, may be beyond the control of the institution, others, such as collaborative learning experiences in the classroom, the amount of contact students have with faculty members, and the way in which student ability is defined, can be influenced by universities. This paper explores the reasons female students give for leaving their ICT courses, and in particular looks at the difference that gender may make in the reasons for attrition. It concludes with recommendations to institutions based on these reasons.

2 Method

The study reported in this paper was part of a broader project investigating attrition. Only those aspects of the project relating to the reasons female students leave their ICT courses are included in this paper. Four Australian universities from different states were involved in the study. Registrars at the four universities identified students who had either transferred from an ICT degree to an unrelated degree, or had left the university altogether, between 2005 and mid 2010. Degrees classified as ICT covered the full spectrum from information systems through to computer science and computer engineering. These 2,868 students were then contacted, requesting their participation in an online survey. Completion of the questionnaire was voluntary and all responses were anonymous.

The online survey comprised 3 main types of questions. The first set of questions captured demographic and background information such as age, gender, marital status, etc. (see Table 1). The second set asked about their early participation in the course, including original enrolment status, if they had attended orientation events, etc. (see Table 1). The third set explored the possible

reasons for participants' withdrawal from their ICT course. This set of questions was presented in four sections. Section 1 asked if their main reason for leaving their degree was due to personal reasons, or if it related to something about the course, or if it was a combination of these (see Table 1). Section 2 asked about experiences of the university itself (see Table 2). Section 3 asked about their course including items relating to academic preparedness, the way the course was taught and run, and aspects of the teaching environment (see Table 3). Section 4 asked about life experiences such as chance events, health, finances, etc. (see Table 4). The items in sections 2, 3 and 4 were presented as negative statements describing possible reasons for attrition (e.g. 'I lost my job') and respondents were asked to rate their agreement with each statement on a 5-point Likert scale ranging from 'Strongly Disagree' to 'Strongly Agree'.

3 Findings and Discussion

Approximately 10% of letters and emails to potential participants were unable to be delivered due to address changes. A total of 154 ex-ICT students (18.8% females and 81.2% males) completed the survey, giving a response rate of 6% for those students who were able to be contacted. The relatively small number of females (29) is consistent with the numbers studying ICT at the universities involved (DEEWR, 2011a), and with the literature on female participation in tertiary ICT education in Western countries (Cory et al. 2006, Lewis et al. 2007). The female respondents' individual characteristics are shown in Table 1 (note: all percentages are percentage of those females who responded to the question).

The majority of the female participants had studied full time (69%) and all were domestic students (100%) while more than a third had been working over 20 hours per week (36%) and caring for dependent children (21.5%). The female participants were predominantly school leavers (58.6%) and this may partially explain their attrition as they may have lacked sufficient maturity to undertake an ICT degree. However 37.8% of female ex-students were in the 20 and over age range when they enrolled, so maturity should not have been an issue. Interestingly only 61.5% of females had enrolled in ICT as their first choice. It is not unexpected that people might leave a non-preferred degree, which would in part explain the female attrition. Similarly it was the first attempt at university study for only 69.2% of females, meaning that a significant proportion of females had either attempted or already completed a previous degree, enrolled in an ICT degree and then left.

The majority of participants had attended orientation activities (73.1%), but only 16.7% had attended functions organised by their school. Many of the students who had not attended functions indicated that either none were organised, or that they were not aware of any. Most of the female students who left their degree had been enrolled in IT (64.3%) and were critical of the course content. Several students mentioned the emphasis on programming and the expectation of prior knowledge as contributors to their decision to abandon their ICT degree.

Student Characteristics	Females	% of females
Age: Under 18	7	24.1
Age: 18	10	34.5
Age: 19	1	3.4
Age Range: 20 to 25	5	17.2
Age Range: 26 to 35	2	6.9
Age Range: 36 to 45	3	10.3
Age Range: 46 to 55	1	3.4
Full-time	20	69.0
Part-time	9	31.0
Domestic	26	100.0
International	0	0.0
Degree First Choice: Yes	16	61.5
Degree First Choice: No	10	38.5
ICT First Degree: Yes	18	69.2
ICT First Degree: No	8	30.8
Attended Orientation: Yes	19	73.1
Attended Orientation: No	9	31.0
Attended Functions: Yes	4	16.7
Attended Functions: No	20	83.3
Enrolled Degree: CS	5	17.9
Enrolled Degree: EE	0	0.0
Enrolled Degree: IT	18	64.3
Enrolled Degree: IS	4	14.3
Enrolled Degree: SE	1	3.6
Enrolled Degree: CE	0	0.0
Hours Worked p/w: 0-10	9	36.0
Hours Worked p/w: 10-20	7	28.0
Hours Worked p/w: 20-30	4	16.0
Hours Worked p/w: 30-40	3	12.0
Hours Worked p/w: 40+	2	8.0
Marital Status: Single	18	64.3
Marital Status: Partner no Child(ren)	4	14.3
Marital Status: Single with Child(ren)	1	3.6
Marital Status: Partner with Child(ren)	5	17.9
Dropped Course: Personal Reasons	3	10.3
Dropped Course: The Course	4	13.8
Dropped Course: Both Personal and Course	22	75.9

Table 1: Individual characteristics of respondents by gender

3.1 Reasons for Attrition

Participants were initially asked if their **main** reason for leaving their degree was due to personal circumstances, due to the course itself, or a combination of both. The majority of female respondents (75.9%) indicated that both personal and course issues had influenced their decision. For example:

"Pressures of changes in workplace increasing work hours beyond what I could fit studies around. The tutors did not answer most of the technical questions I had regarding the course" Female, 26, InfoSys.

Personal reasons alone were the cause for only 10% of the female participants. For example:

"The lack of financial aid which caused great stress and led to illness" Female, 24, CompSci.

While 13.8% indicated that the main reason was course related. For example:

“Course content wasn't practical nor business focussed enough. Where content overlapped with real on the job experience, staff were inflexible and unwilling to award credit...” Female, 18 CompSci.

Participants were then asked to respond to a series of 5-point Likert scales which presented many common reasons for attrition. Table 2 below presents the responses to reasons for attrition that relate to the university experience. The most frequent response was that female students could not get help when they needed it (31.4%). Other reasons included there being too many distractions preventing them from concentrating on their studies (24.1%) and the challenge of organising a timetable with no clashes (20.6%).

University Experience Reasons	N.	SD %	D %	N %	A %	SA %
Lack of help when needed	29	13.9	41.4	10.3	24.1	10.3
Distractions stopped me concentrating on study	29	17.2	41.4	17.2	24.1	0.0
Difficulties organising a suitable timetable	29	24.1	41.4	13.8	17.2	3.4
No opportunities to socialise	29	17.2	31.0	37.9	13.8	0.0
University staff were not friendly	29	17.2	51.7	17.2	10.3	3.4
University facilities were inadequate	29	31.0	34.5	24.1	10.3	0.0
Evening classes posed a security risk	29	34.5	31.0	24.1	10.3	0.0

Table 2: Reasons for attrition: university experience (SD = Strongly Disagree to SA = Strongly Agree)

The issue of least concern was the possible security risk associated with attending evening classes. Although security concerns are mentioned in the literature as a reason for attrition (Marginson et al., 2010) at 10.3% it does not appear to have been a major factor for females in this study.

The next set of reasons for attrition was associated with the course experience and is shown in Table 3. The most frequent response to the reasons relating to the course experience was that classes were boring (51.7%) and many females also found the pace of teaching too fast (41.3%).

In a recent Australian survey of over 30,000 students, ICT students were found to have the lowest levels of academic challenge, higher order thinking and enriching educational experiences of all disciplines considered (ACER 2010). The results of the current study reflect a sense that much ICT teaching may be boring because of its focus on transferring content knowledge at a rapid rate rather than making use of constructivist approaches; this is contributing to attrition.

Consistent with perceptions that ICT teaching can be boring, female participants also frequently showed agreement with reasons relating to the balance between application and theory: lack of workplace focus (42.9%), lack of practical applications (39.3%) and lack of business focus (35.7%). Females also saw the courses as too theoretical (28.5%).

Course Experience Reasons	N.	SD %	D %	N %	A %	SA %
Teaching						
Classes were boring	29	10.3	17.2	20.7	31.0	20.7
Pace was too fast	29	13.8	24.1	20.7	24.1	17.2
Teachers didn't explain exercises	29	10.3	31.0	24.1	24.1	10.3
Not encouraged to do well by teachers	28	14.3	35.7	25.0	21.4	3.6
Teachers were not prepared	29	20.7	51.7	20.7	3.4	3.4
Teachers were out of date	29	13.8	55.2	27.6	3.4	0.0
Harsh, confrontational teaching methods	29	10.3	51.7	37.9	0.0	0.0
Course						
Course lacked workplace focus	28	7.1	17.9	32.1	28.6	14.3
Course lacked practical applications	28	3.6	39.3	17.9	25.0	14.3
Course too mathematical	28	17.9	25.0	21.4	25.0	10.7
Course lacked business focus	28	10.7	25.0	28.6	28.6	7.1
Course was too theoretical	28	10.7	32.1	28.6	21.4	7.1
Poorly structured course	28	10.7	28.6	35.7	21.4	3.6
Too many assignments	28	7.1	35.7	35.7	21.4	0.0
Focus on individual activities rather than groups	28	14.3	39.3	32.1	10.7	3.6
Teaching environment						
Didn't feel I fitted in	27	18.5	14.8	18.5	29.6	18.5
Environment didn't suit my learning style	29	17.2	37.9	6.9	24.1	13.8
Environment unwelcoming	29	15.1	38.2	21.1	17.1	8.6
Course was too competitive	28	17.9	32.1	39.3	7.1	3.6
Preparedness and other issues						
Course didn't meet my expectations	28	7.1	14.3	14.3	35.7	28.6
Didn't enjoy classes	27	11.1	7.4	22.2	44.4	14.8
Didn't understand concepts	28	7.1	10.7	25.0	35.7	21.4
Results were disappointing	28	3.6	28.6	17.9	35.7	14.3
Didn't understand terms used	28	14.3	28.6	14.3	32.1	10.7
Didn't have the expected background knowledge	28	17.9	17.9	21.4	32.1	10.7
Didn't make friends with classmates	26	11.5	23.1	26.9	30.8	7.7
I felt it was unacceptable to be smart	28	42.9	42.9	10.7	3.6	0.0

Table 3: Reasons for attrition: course experience (SD = Strongly Disagree to SA = Strongly Agree)

ICT courses in Australia have the lowest proportion of students undertaking internships (ACER 2010), and a study by Koppi et al. (2010) noted that ICT graduates in the workplace have recommended that students receive more industry related learning. Weng et al. (2010) also called for an increased focus on solving business

problems. The following quote reflects a common sentiment among students:

“Degree simply wasn't what I wanted. Realised after I started it. Although I love IT and always thought I'd study it, I decided a degree combined more with business would be more beneficial” Female, 18, IT.

Issues associated with the teaching and learning environment were also considered important: some females felt that the teaching environment did not suit their learning style (37.9%), or was not welcoming (25.7%) and 48.1% felt that they did not belong. Barker et al.'s (2009) study of predictors of intention to persist in computer science found that when students perceive the workload as being too heavy they are less likely to pursue the major. While this influenced some students (21.4%) it was not the major issue.

Almost two thirds of the female participants also noted reasons such as the course not meeting their expectations (64.3%) and not enjoying classes (59.2%). These sentiments are relatively general and could be associated with a variety of other more specific reasons discussed in this section.

More than half the female students felt that they did not understand the concepts (57.1%), and many felt they did not understand the terms used in the course (42.8%) or did not have the expected background knowledge (42.8%). For example:

“I didn't have the expected background knowledge; the courses were definitely geared towards those with more pre-existing knowledge.” Female, 18, IT.

Having the expected background for ICT studies has been identified in previous research as an important predictor of attrition (Barker et al. 2009).

As indicated earlier, assumptions of prior experience and ability could be modified to prevent the exclusion of those who may have an aptitude for ICT without having spent every waking minute of their teens using computers. This myopic focus on computing (Margolis & Fisher 2002) underpins certain expectations built into the curriculum which are detrimental to those who do not fit the geek stereotype. This issue is explored further below in relation to different types of students.

The social aspect of study also received attention with one third of females (38.5%) agreeing that they didn't make friends with classmates. This was also identified by Barker et al. (2009), who found that levels of student-to-student interaction were perceived as 'unfavourable' by the computer science students in their study, and they recommended that faculty focus on incorporating activities that support interaction. This issue can be addressed in both the nature of the course and in the teaching approaches used.

The responses to possible reasons for attrition that relate to the lives of the students are shown in Table 4 below. More than half of the female participants felt that they had picked the wrong degree (62.9%). This sentiment implies a lack of interest and engagement with the degree content, but could also be associated with a variety of other more specific reasons that are discussed in this section.

Life Experience Reasons	N	SD %	D %	N %	A %	SA %
Picked the wrong degree	27	11.1	11.1	14.8	29.6	33.3
Attending university was too expensive	27	22.2	29.6	22.2	14.8	11.1
Conflicts with my work commitments	26	30.8	34.6	11.5	15.4	7.7
Distance made travel to university difficult	27	25.9	33.3	18.5	14.8	7.4
Travel to university was difficult because of transport	27	25.9	33.3	18.5	11.1	11.1
Timetable didn't fit my work commitments	26	11.5	38.5	30.8	7.7	11.5
I couldn't get financial aid	27	25.9	37.0	22.0	0.0	14.8
My family didn't help me to study at home	27	29.6	33.3	22.2	14.8	0.0
My partner or I got pregnant.	27	44.4	18.5	29.6	3.7	3.7
University study wasn't as important as socialising	27	29.6	48.1	14.8	7.4	0.0
Death, serious illness or accident in the family	27	51.9	25.9	14.8	7.4	0.0
I missed my family	27	25.9	40.7	25.9	7.4	0.0
I lost my job	26	50.0	30.8	15.4	0.0	3.8
Serious illness or accident	27	37.0	33.3	25.9	0.0	3.7
Difficulties living at home	27	40.7	33.3	22.2	3.7	0.0
Living away from home was too difficult	27	18.5	25.9	51.9	3.7	0.0
My timetable didn't fit with the transport timetable	27	25.9	37.0	33.3	3.7	0.0
Difficulties living in student accommodation	27	18.5	25.9	55.6	0.0	0.0

Table 4: Reasons for attrition: students' lives (SD = Strongly Disagree to SA = Strongly Agree)

Financial pressures are of concern to students in all disciplines, and a major predictor of attrition (Bennett 2003, Cabrera et al. 1993). ICT students are no different in this respect. The cost of university education influenced many of the participants. It was considered too expensive by 25.9% of females while 14.8% agreed or strongly agreed that they couldn't get financial aid. Conflicts with work commitments were also a common issue; 23.1% of females agreed that they experienced conflict with work commitments, and 19.2% noted that their study timetable did not fit with their work commitments. Various aspects of travel to university were also found to be problematic for many: distance was an issue for 22.2% of females as was transport availability (22.2%). Factors such as these make it difficult for students to fully engage with their studies and are likely to work in combination with other issues to precipitate attrition

Few female ex-students indicated that they had been affected by serious illness (3.7%), death or illness in the family (7.4%), loss of their job (3.8%) or pregnancy (7.4%).

The results above demonstrate the range of issues that can contribute to female student attrition. It appears that individual students rarely withdraw from their studies for

just one reason. Personal, university and course related issues combine to put pressure on students which may lead to withdrawal. In some cases ex-students feel they have made the decision willingly, but in others they are very conscious of the lack of support received.

3.2 Statistically Significant Gender Differences

Several possible reasons for attrition relating specifically to gender issues were included in the survey. The levels of agreement of the female participants are reported in Table 5 below. Overall, gender issues did not appear to be relatively important to them. Whilst the gender imbalance was certainly noted (62.9% agreement), sexist behaviour from male staff or students was not rated highly as an issue in terms of their withdrawal from the course. For example, only one female participant agreed that male students or staff spoke in a sexist manner, or that male students did not let them participate.

Gender Specific Reasons	N	SD %	D %	N %	A %	SA %
No or few females in class	27	11.1	3.7	22.2	48.1	14.8
In minority in classes	27	18.5	14.8	14.8	48.1	3.7
Male-oriented course content	29	20.7	24.1	27.6	20.7	6.9
Students' sexist behaviour	28	28.6	32.1	32.1	3.6	3.6
Male students stopped me participating	27	25.9	40.7	29.6	3.7	0.0
Male staff not encouraging	27	25.9	33.3	22.2	18.5	0.0
Male staff's sexist behaviour	27	33.3	37.0	25.9	0.0	3.7

Table 5: Responses to gender specific (SD = Strongly Disagree to SA = Strongly Agree)

Some female participants (18.5%) felt that male staff did not encourage them to participate, and 27.6% believed that the course content was male oriented. The general sentiment is captured by the following comment:

“As a female it was quite daunting being a minority in the class but the male students and teachers were in no way deliberately sexist.” Female, 17, IT.

Independent samples t-tests were used to compare the responses of female ex-students to the responses of male ex-students obtained as part of the larger study. Gender was found to have a significant influence on students' agreement with some of the other possible reasons for leaving their ICT course as shown in Table 6.

Females were more likely to believe that they didn't have the expected background knowledge for the course (t=-2.25, p<0.026), didn't understand the concepts (t=-3.82, p<0.001), or didn't understand the meaning of terms used in the course (t=-2.30, p=0.027). Previous research has suggested that female students have no less ability to undertake ICT courses than male students (Beyer et al.,

2003), however, it has been found that female ICT students lack confidence in their ability to achieve their educational goals (Beyer et al. 2003).

Reasons	Females		Males		
	Mean	SD	Mean	SD	Sig.
Distractions stopped me concentrating on study	2.48	1.06	3.06	1.22	0.021
Didn't understand the concepts	3.54	1.17	2.57	1.21	<.001
Lacked the expected background	3.00	1.30	2.41	1.24	0.026
Didn't understand the terms used	2.96	1.29	2.36	1.02	0.027
My results were disappointing	3.29	1.15	2.73	1.10	0.018
I was in the minority in my classes	3.04	1.26	2.41	1.25	0.021
Picked the wrong degree	3.63	1.36	3.02	1.41	0.043

Table 6: Reasons with significantly different levels of agreement between females and males

The findings of this study are consistent with this previous research although it was suggested earlier that there are identifiable reasons for this loss of confidence which can be addressed. Lack of confidence in ability to undertake study in a discipline that is perceived to be challenging is thought to contribute to low enrolment rates of females (Manis et al. 1989), however, questions should be asked about why computing should be challenging and whether this is, again, male modelling of computing (Margolis & Fisher 2002) as a field in which women do not fit. It also appears to contribute to female attrition, preventing female students from accessing the benefits that can flow from an ICT career. Actions that increase confidence should be pursued. These might include mentoring (Cohoon 2001), early exposure to work integrated learning or rethinking the expectations imposed on students by the current design of computing courses whereby students who demonstrate prior knowledge take more complex courses in their first year while those with less knowledge and skills are brought up to the expected level at a less challenging pace (Margolis & Fisher 2002).

Female ex-students were also more likely to say that their results were not as high as they had expected (t=-2.40, p=0.018), and that they felt they had picked the wrong degree (t=-2.04, p=0.043). Previous research has shown that female students who leave ICT degrees tend to have higher grades than males students who do not leave (Strenta et al. 1994), yet they are more sensitive to perceptions that their grades are lower than those they received in high school (Jagacinski et al. 1988). The culture of computing must be highlighted, once more, to explain why this is the case. Seymour and Hewitt (1997, pp241-242) identified a “process of discouragement” which manifested itself in female students: doubting their abilities; having a reduced capacity to deal with setbacks; and being more dependent on reassurance from other people.

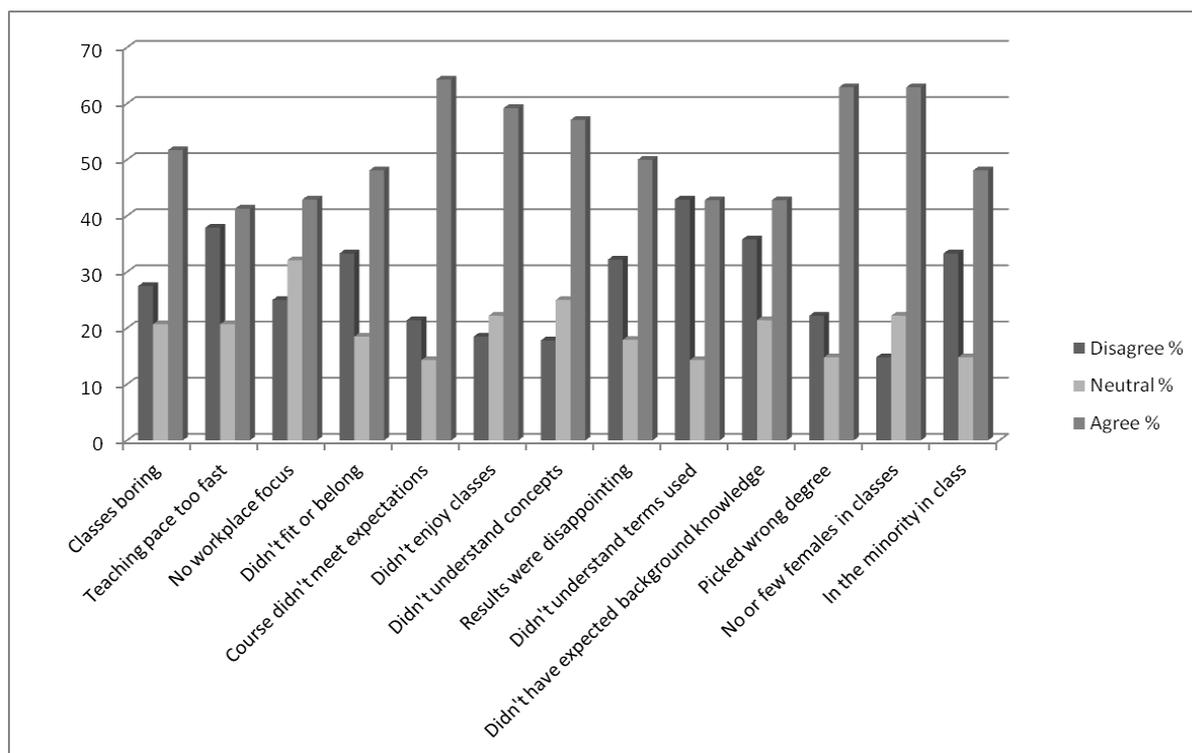


Figure 1: Dominant reasons for leaving ICT courses

Differential attrition of female students in this way is a major loss to the ICT profession, but it is not purely a gender issue, as Strenta et al. (1994) found that in other disciplines, such as science and engineering, where persistence was the same grades were the same.

Unexpectedly, there were no significant differences in response to most of the life issues: female students were not more likely to be affected by issues such as pregnancy or dealing with family illness.

4 Summary and Conclusions

The results presented in the sections above demonstrate the wide range of issues that can contribute to female student attrition. Figure 1 summarises those that were most frequently cited (25% or higher agreement).

Student attrition is an issue of serious concern to universities around the world. It is of particular concern to the field of ICT because of the shortfall of ICT professionals (ACS 2008) as it poses current and future risks for the ICT industry. Not only is there a lack of qualified people in sufficient numbers but, more significantly, there is a lack of qualified women able to contribute their ideas and assist in steering technological developments. This study has attempted to further understand the causes of female attrition from ICT courses by exploring the reasons female students from four Australian universities gave for leaving their ICT courses.

There are many factors that can contribute to the attrition of ICT students, and for many students it is a combination of issues that leads to their withdrawal. Some of the issues identified in this study are beyond the control of universities but many could be mitigated by universities taking appropriate action.

Contrary to commonly held beliefs about women as those most frequently affected by, and expected to manage, serious life events, only a relatively small number of female ex-students had experienced events such as death or serious injury in the family, pregnancy, or loss of their employment, that necessitated their withdrawal. They were also no more likely than males to consider withdrawing due to these issues. It was much more common for the participants to cite reasons associated with the university environment, the teaching of their ICT course, and their inability to combine their studies with other commitments. A theme in issues associated with the university environment was the difficulty in obtaining help when required. Providing greater levels of support during the initial enrolment process, and when students need to make changes to their enrolment to accommodate other challenges in their lives, would address a number of the factors that students have indicated influenced their decision to withdraw. As female students are very much in the minority, and likely to feel isolated, providing ongoing support could be very valuable in alleviating their feelings of not fitting in or belonging to their ICT course and, thereby, increasing their enjoyment of classes.

The course related issues that made a major contribution to female student withdrawal were related to the style of teaching and to the focus of the ICT course. Many female ex-students had found their classes boring, yet they also noted that the pace of teaching was often too fast, and exercises were not explained well. These sentiments have also been expressed by students who continue with their ICT course, resulting in ICT courses being ranked as having the lowest levels of enriching educational experiences and higher order thinking of all courses considered in a survey of over 30,000 students

(ACER 2010). The way in which ICT is taught clearly requires urgent consideration. Recommendations from the ICT education literature include increasing the use of small group class activities (Barker et al. 2009, Powell 2008). Small group activities provide students with opportunities to undertake more active learning, addressing the boredom issue (Schweitzer & Brown, 2007), but also to increase levels of interaction with other students and faculty. Increasing this interaction reduces the likelihood of students feeling disconnected from the teaching and learning environment, and makes it easier for them to ask for support when they need it. These kinds of activities are particularly useful as they help ensure female students feel they are active participants in the class and in their own learning.

In addition to the style of teaching, the balance between application and theory was also of concern. Courses were seen to lack a workplace or business focus and to lack practical application. This finding is not just applicable to students who withdraw; students who have successfully completed their course and obtained work in the ICT industry have also called for more industry related learning (Koppi et al. 2010). Increased use of case based teaching can tie ICT content to application, enabling students to understand the context in which their knowledge will be applied (Mukherjee 2000, Weng et al. 2010). Providing students with an understanding of the social context in which human beings can benefit from ICT may be one of the most important changes to teaching that can be made for all students (Rosser 1990). Better integration of practical and workplace knowledge and skills can also be achieved through providing forms of work integrated learning (e.g. industry related projects or work placements). Team based projects that address problems or opportunities provided by companies, government departments or community organisations enable students to gain professional skills while ensuring that curriculum is aligned with industry needs. Work placements (or internships) are another way to provide students with valuable experience and to strengthen their sense of the relevance of their ICT course. Addressing the perceived lack of workplace focus will lead to committed students who can see where their ICT degree is taking them, possibly providing a greater incentive to work through issues that might be making students consider withdrawing.

Many female students were influenced by a perception that they did not have the expected background knowledge and, as a result, did not understand the terms and concepts used in the course. Previous ICT experience has been found to be an important predictor of attrition (Barker et al. 2009). This issue can be successfully addressed by implementing alternate pathways, so that those students without a strong background take an alternative initial unit in their first year that provides the opportunity to develop the skills and confidence to be successful. This approach has been shown to be particularly valuable in addressing the attrition of female students, as they are more likely to believe that they do not have the necessary background (Powell 2008). Increasing their foundational knowledge would also lead to better results for female students. Gaining higher marks would increase the satisfaction female students

have with the course, encouraging them to believe they have chosen a suitable degree and motivating them to continue with their studies. Other strategies that have had success in improving female student retention include ensuring a gender balance in faculty and providing mentoring (Cohon 2001).

In order to gain further insight into the issues discussed above, it would be useful for future research to contrast these results with responses for ICT graduates. Some issues may not necessarily be institutional or course problems, but relate more to differing student perceptions. Approaches to changing these perceptions could then be explored.

5 Acknowledgements

This research was supported by an ALTC Priority Project grant. The following additional project team members contributed to the survey design: Tony Koppi, Philip Ogunbona and Fazel Naghdy (University of Wollongong); Jocelyn Armarego (Murdoch University); Chris Pilgrim (Swinburne University of Technology); and Paul Bailes (University of Queensland).

6 References

- ACER (2010): Doing more for students: enhancing engagement and outcomes. Australasian Student Engagement Report. http://ausse.acer.edu.au/images/docs/AUSSE_2009_Student_Engagement_Report.pdf. Accessed 6 Apr 2011.
- ACS (2008): The ICT skill forecast project. First report: quantifying current and forecast ICT employment. <http://www.acs.org.au/attachments/ICTSkillsForecastingReportExecSummaryAug08.pdf>. Accessed 1 Apr 2010.
- Bailey, M. and Borooah, V.K. (2007): Staying the course: an econometric analysis of the characteristics most associated with student attrition beyond the first year of higher education. Ulster, Ireland, DELNI.
- Barker, L.J., McDowell, C. and Kalahar, K. (2009): Exploring factors that influence computer science introductory course students to persist in the major. *SIGCSE Bulletin* **41**(2):282-286.
- Barrow, M., Reilly, B. and Woodfield, R. (2009): The determinants of undergraduate degree performance: how important is gender? *British Educational Research Journal* **35**(4):575-597.
- Bean, J.P. (1980): Dropouts and turnover: the synthesis and test of a causal model of student attrition. *Research in Higher Education* **12**(2):155-187.
- Beekhoven, S., De Jong, U. and Van Hout, H. (2002): Explaining academic progress via combining concepts of integration theory and rational choice theory. *Research in Higher Education* **43**(5):577-600.
- Bennett, R. (2003): Determinants of undergraduate student drop out rates in a university business studies department. *Journal of Further and Higher Education* **27**(2):123-141.
- Beyer, S., Rynes, K., Perrault, J., Hay, K. and Haller, S. (2003): Gender differences in computer science students. *SIGCSE Bulletin* **35**(1):49-53.

- Cabrera, A.F., Nora, A. and Castaneda, M.B. (1993): College persistence: structural equations modeling test of an integrated model of student retention. *The Journal of Higher Education* **64**(2):123-139.
- Charles, M. and Bradley, K. (2006): A matter of degrees: female underrepresentation in computer science cross-nationally. In J.M. Cohoon and W. Aspray *Women and information technology: research on under representation*. Cambridge, Massachusetts, MIT Press.
- Cohoon, J.M. (2001): Toward improving female retention in computer science. *Communications of the ACM* **44**(5):108-114.
- Cohoon, J.M. and Aspray, W. (eds) (2006): *Women and information technology: research on under-representation*. Cambridge, Massachusetts, MIT Press.
- Cory, S.N., Parzinger, M.J. and Reeves, T.E. (2006): Are high school students avoiding the information technology profession because of the masculine stereotype? *Information Systems Education Journal* **4**(29):3-13.
- Crisp, G., Nora, A. and Taggart, A. (2009): Student characteristics, pre-college, college, and environmental factors as predictors of majoring in and earning a STEM degree: an analysis of students attending a hispanic serving institution. *American Educational Research Journal* **46**(4):924-942.
- DEEWR (2011a): Students, selected higher education statistics. Canberra, DEEWR.
- DEEWR (2011b): Students, selected higher education statistics (No. RFI 10-324 Roberts).
- e-skills UK (2011), Technology insights 2011: key findings. <http://www.e-skills.com/Research/Research-publications/Insights-Reports-and-videos/Technology-Insights-2011/Technology-Insights-2011-Key-findings/>. Accessed 1 Apr 2011.
- Frieze, C. (2005): Diversifying the images of computer science: undergraduate women take on the challenge! *SIGCSE Bulletin* **37**(1):397-400.
- Gras-Velazquez, A., Joyce, A. and Debry, M. (2009): Women and ICT: Why are girls still not attracted to ICT studies and careers? from http://blog.eun.org/insightblog/upload/Women_and_IC_T_FINAL.pdf. Accessed 3 Apr 2011.
- Hagedorn, L.S. (2005): How to define retention: a new look at an old problem in A. Seidman *College student retention: formula for student success*. Westport, Connecticut, Praeger.
- Hinton, L. (2007): Causes of attrition in first year students in science foundation courses and recommendations for intervention. *Studies in Learning, Evaluation, Innovation and Development* **4**(2):13-26.
- Hovdhaugen, E. (2009): Transfer and dropout: different forms of student departure in Norway. *Studies in Higher Education* **34**(1):1-17.
- ITU (2010), New ITU report Shows Global Uptake of ICTs Increasing, Prices Falling. http://www.itu.int/newsroom/press_releases/2010/08.html. Accessed 3 Apr 2011.
- Jagacinski, C.M., Lebold, W.K. and Salvendy, G. (1988): Gender differences in persistence in computer-related fields. *Journal of Educational Computing Research* **4**(2):185-202.
- Koppi, T., Edwards, S.L., Sheard, J., Naghdy, F. and Brookes, W. (2010): The case for ICT work-integrated learning from graduates in the workplace. *Proc. Australasian Conference on Computing Education*, Brisbane, Australia.
- Kramer, G.L. (2007): Fostering student success in the campus community. San Francisco, Jossey-Bass.
- Lewis, S., Lang, C. and McKay, J. (2007): An inconvenient truth: the invisibility of women in ICT. *Australasian Journal of Information Systems* **15**(1):59-76.
- Lewis, T.L., Smith, W.J., Belanger, F. and Harrington, K.V. (2008): Are technical and soft skills required?: the use of structural equation modeling to examine factors leading to retention in the CS major. *Proc. International Workshop on Computing Education Research*, Sydney, Australia.
- Logan, K. and Crump, B. (2007): The value of mentoring in facilitating the retention and upward mobility of women in ICT. *Australasian Journal of Information Systems* **15**(1):41-58.
- Manis, J., Sloat, B.F., Thomas, N.G. and Davis, C.S. (1989): An analysis of factors affecting choices of majors in science, mathematics and engineering at the University of Michigan. Michigan, University of Michigan.
- Mann, S. and Robinson, A. (2009): Boredom in the lecture theatre: an investigation into the contributors, moderators and outcomes of boredom amongst university students. *British Educational Research Journal* **35**(2):243-258.
- Marandet, E. and Wainwright, E. (2010): Invisible experiences: understanding the choices and needs of university students with dependent children. *British Educational Research Journal* **36**(5):787-805.
- Marginson, S., Nyland, C., Sawir, E. and Forbes-Mewett, H. (2010): *International student security*, Melbourne, Cambridge University Press.
- Margolis, J. and Fisher, A. (2002): *Unlocking the clubhouse: women in computing*, Cambridge, Massachusetts, MIT Press.
- Marks, G. (2007): Completing university: characteristics and outcomes of completing and non-completing students. Australian Council of Educational Research. http://research.acer.edu.au/lsay_research/55. Accessed 30 Nov 2009.
- Mukherjee, A. (2000): Effective use of in-class mini case analysis for discovery learning in an undergraduate MIS course. *Journal of Computer Information Systems* **40**(3):15-23.
- Nora, A., Barlow, E. and Crisp, G. (2005): Student persistence and degree attainment beyond the first year: the need for research in A. Seidman *College student retention: formula for student success*. Westport, Connecticut, Praeger.

- Powell, R.M. (2008): Improving the persistence of first-year undergraduate women in computer science. *SIGCSE Bulletin* **40**(1):518-522.
- Price, D., Harte, J. and Cole, M. (1992): Student progression in higher education: a study of attrition at Northern Territory University. Canberra, Australian Government Publication Service.
- Rosser, S. (1990): *Female-friendly science: applying women's studies methods and theories to attract students*. New York, Pergamon Press.
- Schweitzer, D. and Brown, W. (2007): Interactive visualization for the active learning classroom. *SIGCSE Bulletin* **39**(1):208-217.
- Seidman, A. (ed.) (2005): *College student retention: formula for student success*. Westport, Connecticut, Praeger.
- Seymour, E. and Hewitt, N.M. (1997): *Talking about leaving: why undergraduates leave the sciences*. Boulder, Colorado, Westview Press.
- Stater, M. (2009): The impact of financial aid on college GPA at three flagship public institutions. *American Educational Research Journal* **46**(3):782-815.
- Strenta, A.C., Elliott, R., Adair, R., Matier, M. and Scott, J. (1994): Choosing and leaving science in highly selective institutions. *Research in Higher Education* **35**(5):513-547.
- Telecompaper (2010): Number of ICT workers in Germany at record levels. <http://www.telecompaper.com/news/number-of-ict-workers-in-germany-at-record-levels-bitkom>. Accessed 19 Oct 2010.
- Tinto, V. (1975): Dropout from higher education: a theoretical synthesis of recent research. *Review of Educational Research* **45**(1):89-125.
- Tinto, V. (1993): *Leaving college: rethinking the causes and cures of student attrition*. 2nd edn, Chicago, University of Chicago Press.
- Weng, F., Cheong, F. and Cheong, C. (2010): Modelling IS student retention in Taiwan: extending Tinto and Bean's model with self-efficacy. *ITALICS* **9**(2):97-108.

Work Integrated Learning Rationale and Practices in Australian Information and Communications Technology Degrees

Chris J Pilgrim

Centre for Computing and Engineering Software
Systems, Faculty of ICT
Swinburne University of Technology
PO Box 218, Hawthorn, 3122, Victoria
cpilgrim@swin.edu.au

Tony Koppi

Faculty of Informatics
University of Wollongong
Northfields Ave, Wollongong, 2522, NSW
tkoppi@uow.edu.au

Abstract

To obtain a better understanding of WIL rationale and practices in Australian ICT degrees, a survey of managers and educational leaders of ICT was undertaken. These survey results were analysed and informed by discussions at a forum of ICT educational leaders. Results indicate that WIL practices are broad with a wide range of internal (university) and external (industry) combinations to provide the student with appropriate professional experience. The majority of respondents indicated that their curricula are industry relevant, and that they offer an industry-linked final year project. Virtual or simulated work experiences also seem to be commonly practiced. The range of options is influenced by local context, staff approaches and resource availability. The majority of universities regard WIL as important and beneficial and apparently have practices that provide for industry contribution to the curriculum even though this may not be obvious to graduates in the workplace. Support provided to students for an industry placement is variable. Success measures of placements are that students have improved understanding of professional responsibility and have gained a variety of work perspectives. That the student is employable as a consequence is not seen as very important. There appears to be a tension between desired outcomes from academia and industry including those of 'work readiness' and lifelong learning. It seems that the range of options provided by universities need to be recognised by all stakeholders as contributing to the development of an ICT Professional.

Keywords: Work integrated learning, professional practice, student experience, industry, academia.

1 Introduction

A survey of management and educational leaders of Information and Communication Technology (ICT) departments and schools from Australian universities was carried out as part of a recent project funded by the Australian Learning and Teaching Council (ALTC) project (Ogunbona, 2009). One of the key aims of the project was to investigate the lack of real-world

experience that was strongly felt by recent ICT graduates in the workplace as reported in a previous related ALTC project (Koppi and Naghdy, 2009). The previous project found a significant mismatch (88%) between what the graduates in the workplace considered important abilities for their work and how they perceived universities had prepared them for those abilities. The aim of the survey of management and educational leaders was to obtain an understanding of the representative views and practices of Work Integrated Learning in ICT in Australian universities. The findings from the survey complement those of the previous ALTC study which focused on the perceptions of recent graduates in the workplace therefore providing a comprehensive picture of WIL practices from each perspective.

The term 'Work Integrated Learning' (WIL) is now commonly regarded as an umbrella term that covers a "range of approaches and strategies that integrate theory with the practice of work within a purposefully designed curriculum" (Patrick et al, 2009). An alternative definition of WIL is "the process whereby students come to learn through experiences in educational and practice settings and reconcile and integrate the contributions of those experiences to develop the understandings, procedures and dispositions, including the criticality and reflexivity, required for effective professional practice" (Billett, 2011). The key characteristics drawn from these definitions are that WIL involves a range of models of learning experiences with the common aim of developing student's professional capabilities and knowledge of the workplace to equip them for professional practice.

The benefits that WIL brings to all stakeholders, including students, universities, industry and the economy have been well documented (e.g., Poppins, and Singh, 2005; Pauling and Komisarczuk, 2007). WIL provides students with an opportunity to test the theoretical knowledge learnt at university and to put it into action in the "complex and pressurized environment of the real professional world" (Bates et al, 2007). Billett (2011) in his ALTC report on Integrating Practice-Based Experiences identified several reasons for integrating work-based learning experiences into the higher curriculum including learning about an occupation, extending the knowledge learnt in university settings, and building the capacities required to engage in and be an effective professional practitioner. WIL provides graduates with significant salary advantages with a reported median starting salary AUD\$13,000 higher for those with previous work experience in computing (GCA,

2010). University lecturers in Australia have identified Industry-Based Learning as the single best feature of their degrees because it realized the alignment of their programs to industry (Smith et al, 2008). Similarly, the previous related ALTC project (Koppi and Naghdy, 2009) found that ICT graduates in the workplace strongly believe that university courses should contain some form of work-integrated learning and also that ICT employers believe that students need more work placements to gain industry experience. The government also recognises the value of WIL to the economy with the Minister for Employment Participation stating “By integrating practice and theory, students develop those important ‘softer’ skills greatly valued by employers, such as team work, self-management and initiative. Students are able to make an immediate and meaningful contribution to increasing productivity and prosperity—for industries, businesses and the nation as a whole” (O’Connor, 2008).

Whilst the University sector recognises and acknowledges the significant benefits of the objectives of WIL there remains some questions regarding the actual work readiness and professional preparation of graduates. For example, the related ALTC project (Koppi and Naghdy, 2009) identified deficiencies in the workplace readiness of new graduates particularly in relation to the development of essential generic skills such as interpersonal and professional communications, business awareness and problem-solving abilities. Likewise, the Business Council of Australia claim that graduates still lack the essential attributes especially in leadership, teamwork and communication, and that “Universities were failing to heed the call” (Hare, 2011; BCA, 2011).

This paper reports on a survey of academic leaders of ICT departments and schools in Australian universities regarding the rationale and practices of Work Integrated Learning. The results of this survey are contrasted with the results of a related project that surveyed recent graduates in the workplace and ICT employers. The perspectives and perceptions reported in this paper will assist in the development of a nationally coordinated approach to WIL in ICT educational programs that will benefit all stakeholders including students, employers and universities.

2 Survey and Analysis Methods

2.1 Survey Groups

The aim of the survey designed by the project team was to obtain an understanding of the representative views and practices of WIL in ICT in Australian universities. The first targeted group was the Heads of ICT organisational units at the Australian universities who were members of the Australian Council of Deans of ICT. A series of four approaches was used in order to obtain their participation in the data-gathering exercise: a paper-based survey was mailed to each university representative on the Council; a fortnight later an emailed survey was sent to the same people; a telephone follow up was undertaken two weeks later and finally they were sent an invitation to complete the survey online. As a result of these efforts a total of 22 completed surveys were received from 18 universities (a few ICT heads had distributed the survey to other ICT heads internally).

The second group to complete the same survey was the Associate Deans for Learning and Teaching (or their equivalent) in ICT at a forum of 36 attendees representing 25 universities, and 30 completed surveys were received.

One workshop session at the forum was concerned with WIL issues and the recorded discussions were also used to inform the project. The total of 52 completed surveys and forum deliberations are considered as representative of WIL views and practices amongst Australian ICT academia.

2.2 Survey Analysis

The survey consisted of a number of questions to be rated on a 5-point Likert Scale where a tick was sufficient to indicate the response and an option to provide further comments. Entries to survey tick boxes were compiled to provide quantitative data. Free text entries were read repeatedly to enable the coding and categorisation of responses which were then counted to enable quantitative comparisons. This qualitative data analysis method was informed by the work of Boyatzis (1998), and Bogdan and Bicklen (2002).

2.3 Forum Discussions

Participants at the forum were broken into six small groups to facilitate workshop discussions on a broad range of WIL issues in ICT. Their deliberations were summarised on paper by each group, collected and later compiled. Plenary discussions were summarised and typed for the whole group to see on screen and edit at the time.

3 Findings

The significant benefits of WIL has incentivised universities to develop and implement a range of models of WIL extending from the traditional work experience placement or internship programs to innovative virtual or simulated WIL experiences. The range of models have also been acknowledge by the government with O’Connor (2008) noting that WIL comes in many different forms including “research, internships, studying abroad, student teaching, clinical rotations, community service or volunteer work, industry attachments or placements, sandwich programs, and professional work placements”. Boud and Symes (2000) regard all models of WIL, including those that occur in a workplace, in the community, within the university, and real or simulated, as valid “as long as the experience is authentic, relevant and meaningfully assessed and evaluated” (Boud and Symes, 2000).

12 month paid industry placement	16
6 month paid industry placement	17
Industry-linked final year project	43
Unpaid internships	23
Industry relevant curricula	44
Virtual or simulated work experience	22

Table 1: Survey results showing the WIL opportunities available to students

Table 1 shows the tick-box results from the 52 respondents regarding the kinds of WIL opportunities

available to students at their institution. Respondents may have ticked more than one box. Most respondents indicated that their curricula are industry relevant and that the final year project is somehow linked to industry.

Virtual or simulated work experiences seem to be a common practice. Forum attendees were overwhelmingly in support of WIL models that provided authentic work experience for students. However the forum participants also discussed alternative opportunities for students unable to attend a workplace (such as by means of a placement). For those students, a virtual or simulated experience may be the next best option. Unpaid internships were also indicated by a similar number of respondents, and paid industry placements were the least available to students and there was little difference between the frequency of 6- or 12-month placements.

Survey respondents also had the opportunity to specify other options available to students or comment on the tick-box options, and these included:

- Funded placements through WIL scholarships
- Placements vary from a few weeks to about three months, and may be part-time, e.g., 2.5 days/week or a flexible 100 hours during the course
- Paid internships in research organisations
- Guest teaching by industry professionals
- Assignments requiring interviews and interaction with ICT professionals in industry
- Industry certified courses (e.g., CISCO)

The range of work integrated learning opportunities appears broad from a national perspective but the options at the local level will depend upon the university location (metropolitan or rural), local context, staff approaches and resource availability. The forum participants agreed that a range of models was required in order to provide the flexibility to accommodate the diversity of student capabilities, motivations and interests as well as different university resourcing models and priorities.

3.1 Local WIL Practices and Support: Survey Responses

Table 2 shows local practices within ICT schools or departments. Responses range from Strongly Disagree (SD) to Strongly Agree (SA) with the proportion (%) of entries per box and ranked according to the strength of agreement (A + SA) with the given statements.

According to the academic staff that completed the survey, the majority of universities have practices that provide for industry contribution to the curriculum. However, when ICT graduates in industry were asked about their curriculum and workplace preparation, the majority stated that an area in need of improvement concerned industry involvement in the curriculum (Koppi and Naghdy, 2009). This same study also found that ICT employers desired greater input to the curriculum, and is consistent with the wishes of ICT employers found in a survey by Hagan (2004). Greater industry and university liaison over the curriculum would appear to be a challenge.

The majority of universities regard WIL as a key feature of ICT degrees and actively encourage students to undertake a placement and will only approve such a placement if it provides the student with an appropriate

learning experience. A little over half of the universities actively find and manage placements and believe that industry should support the management of such programs. About half of school or department academic staff provide support for industry engagement with WIL although less than half provide support for students with an induction program. A similar proportion emphasise the development of generic skills during WIL experiences. A minority of universities provide a high level of resources for WIL.

	SD	D	N	A	SA
Seeks industry input into curriculum design		2	8	46	44
Has policies that require industry input into curriculum design		4	12	46	38
Requires WIL to provide an appropriate learning experience		10	14	36	40
Actively encourages students to undertake a placement		10	18	37	35
Regards WIL as a key feature of the ICT degrees		15	21	42	21
Actively manages IBL or internship placements	2	24	12	38	24
Believes that industry should financially support WIL programs	2	8	31	37	22
Finds IBL or internship placements for students	6	23	19	30	23
Has academic staff who support WIL activities		30	22	32	16
Emphasises the development of generic skills rather than competencies in WIL		12	40	40	8
Has an induction program for students entering placements	4	26	24	30	16
Provides a high level of resourcing for WIL	8	38	24	22	8

Table 2: Local university WIL practices and support

3.2 Local WIL Practices and Support: Forum Discussions

While recognising that industry placements provide benefits to students, it was acknowledged that they were not available to all for a variety of reasons. It was clear that placements were more readily available for the better students and there was uncertainty about whether or not the weaker students would benefit and that hard evidence was lacking. From the survey of ICT graduates in industry, Koppi and Naghdy (2009) reported that the graduates found that industry placements gave them a better appreciation of the relevance of university courses and provided them with a framework for their studies upon returning to university. It could be argued that these experiences are precisely what would benefit weaker students the most.

University location and the economic climate affect placement opportunities. Regional universities offering ICT courses may not be conveniently close to industries that could provide placements and in economic downturns, even metropolitan universities may find it difficult to place students in industries that could be shedding staff. Established relationships between universities and industry partners provide the most stable

circumstances for placements even though they may be restricted to relatively few students.

While universities may encourage placements in ICT jobs, students themselves may not avail themselves of such opportunities because they already have established part-time non-ICT jobs that are necessary to maintain themselves through university. This is supported by recent research that found half of Gen Y students in full-time study also have paid jobs (AMP-NATSEM 2007).

It was observed that the development of generic skills (sometimes called 'soft skills') in industry employment may occur in any workplace context and not necessarily ICT employment. This same observation was made by ICT graduates in industry who reported that they learned generic skills such as negotiation with clients, communication and teamwork in a variety of workplaces (Koppi and Naghdy, 2009). It may be that the Gen Y students in paid prolonged employment in any business may be developing generic skills (or employability skills) to at least the same level as they would in an ICT placement, especially where ICT positions are unavailable.

The point was also made that university administration of placements was a burden that demanded resources and constant effort to maintain relationships with relevant industries.

3.3 Success Measures of Placements

Table 3 shows the proportional responses (%) of academic staff to the statement: 'The success of an Industry-Based Learning or internship placement is judged when the student:'. Responses range from Strongly Disagree (SD) to Strongly Agree (SA) with ranking according to the strength of agreement (A + SA) with the given statements.

	SD	D	N	A	SA
Has improved understanding of professional responsibility		2	9	45	45
Gained a variety of work perspectives		4	7	62	27
Has completed work tasks as required		2	9	72	17
Has gained new technical skills and competencies		6	36	47	11
Did not disrupt normal company operations	7	15	37	30	11
Is now employable	7	22	35	26	11
Added value to the company's profitability		38	40	22	

Table 3: Success Measures of Placements

The students who achieved an improved understanding of professional responsibility and gained a variety of work perspectives are regarded as having achieved the strongest learning outcomes. These outcomes would help with employability and would be difficult to obtain by any other means. However, these outcomes are generic and could be acquired from a number of professional employment situations; the context would determine ICT relevance. One of the benefits of workplace experience is employability, as many employers require such experience and 'work readiness' even in recent graduates

(Forth and Mason, 2003; Pauling and Komisarczuk, 2007; Kennan et al., 2008), however this outcome is apparently of limited concern to most academic staff who did not rate the outcome of the student being employable as very high (Table 3).

Completion of work tasks is a strongly desired outcome but these are not necessarily related to gaining new technical skills and competencies because the proportion of responses to these two outcomes is different. On balance, not disrupting normal company operations is seen as a success but adding financial gain to the company is generally not.

Additional comments made by survey respondents with respect to success measures include the attainment of analytical skills, better interpersonal skills, more realistic views about the workplace and work politics, and improved self-organisation. In addition to not disrupting company operations, it was noted that the students should not harm university-industry relations.

Free text responses indicated that there is a broad range of placements with a variety of outcomes. Students may be on placements for lengthy full-time (6 months or more) or part-time periods, or just for a few weeks obtaining some form of relevant work experience with a report to prove it was carried out. Other placements are based on a learning plan with specific learning activities and outcomes.

The forum participants unanimously agreed that WIL is beneficial in developing certain 'professional attributes' in students and would improve student's employment outcomes. The value of WIL beyond the direct employment benefits was discussed at the forum with some participants noting that employment outcomes should not necessarily be seen as the primary goal of university education. The tension between teaching theory and vocational practice when designing of curriculum for ICT degree programs has resulted in some employers believing that "universities are not interested in meeting industry requirements" (Koppi and Naghdy, 2009). An academic goal is to develop rounded graduates with life-long learning skills whereas some industries desire graduates who are trained in the contemporary tools and techniques used in current corporate and industry environments (Shoikova & Dwishev, 2004).

Several respondents commented on this tension between academic and industry regarding placements and these have been summarised in Table 4.

Many academic staff consider placements as essential but recognise the limitations without university and industry support. This is particularly so for regional universities where there may be insufficient local places available to meet student demand. Several participants mentioned that an effective placement strategy should be at a national level managed by stakeholders including universities (especially administrative support), the Australian Computer Society, Engineers Australia, the Australian Information Industry Association, and government. The students themselves are key stakeholders and it is unfortunate that the value of placements to attain industry experience is often not appreciated until after graduation, as noted by survey respondents and found by Koppi and Naghdy (2009) in their survey of ICT graduates in industry.

Industry	Academia
Work ready	Academia ready
High-level communication skills	Balance between communication skills and expression of knowledge
Profit-making environment	Knowledge-making environment
Want high-performing students	Have students with a wide range of abilities
Time for effective student contribution	Time away from formal teaching
Variable demand for students	Constant requirement for places
Expect universities to provide resources	Would like more industry contribution

Table 4: Tensions between Industry and Academia over Placements

4 Discussion

The results of the survey and forum discussions indicate that there is a range of rationale and practices for WIL in Australian universities. Universities recognise the educational and employment benefits of WIL and generally regard WIL as a key feature of ICT programs. Resourcing for WIL varies across the sector possibly influencing a variety of models of WIL that extend from the traditional work experience placement to new virtual or simulated WIL experiences. Universities appear to advocate for more flexibility in WIL models to meet the diversity of student capabilities and interests, including international students and those students with significant part-time jobs. Universities also indicated that appropriate models of WIL are required to suit different university resourcing models and priorities.

The results indicate that the most prevalent WIL models were the ‘Industry-Linked Final Year Project’ and ‘Industry Relevant Curricula’. These models may be classified as ‘internal’ using a continuum from the traditional ‘external’, industry-based WIL experiences such as work experience placements and internships to ‘internal’, university-based experiences such as project work, case studies and experiential learning opportunities. Fewer universities provided traditional 6 month or 12 month paid industry-based learning placements which has been the WIL model traditionally preferred by the ICT industry (Mather, 2010).

The use of Industry-Linked Final Year Projects as the key method of providing a WIL experience in ICT degrees is endorsed by the Australian Computer Society in their Accreditation Guidelines (ACS, 2009) which state that programs will “include a capstone unit in the final year to allow an assessment of the program objectives.” The guidelines contain a Policy on Capstone Units (Appendix 3) that indicate dual objectives for capstone units:

1. Integrate the skills and knowledge developed throughout the program;
2. Provide a structured learning experience to facilitate a smooth transition to professional practice or further study in the discipline.

The Policy does not provide details of the types of learning experiences that would be appropriate to achieve these objectives apart from a statement regarding the need for “authentic learning experiences in relation to its intended professional outcomes”.

The issue of the authenticity of learning experiences is central to the success of WIL programs; however agreement on what makes a WIL learning experience authentic appears to be split between academic and industry views. The results of the survey indicate that universities believe that a successful WIL experience provides students with an improved understanding of professional responsibility and the attainment of generic skills. The forum discussions however raised concerns from universities that industry also stresses the need for ‘work ready graduates’ (Mather, 2010) possibly at the expense of a more holistic education with a focus on life-long learning. This issue was described as an ‘expectations gap’ in the ALTC WIL Project (Patrick et al, 2009) which recommended a “stakeholder integrated approach to the planning and conduct of WIL based on formalised, sustainable relationships and a common understanding of the procedures and commitment required by all those involved.”

An approach to develop a shared understanding regarding the authenticity of the range of learning experiences for WIL is required in order to achieve industry acceptance and recognition of innovative internal and virtual models of WIL.

The movement towards outcomes-based education in engineering education may provide a way forward to achieving a common understanding of the value of the full spectrum of WIL models. The focus on educational outcomes rather than learning methods is now endorsed by both universities and industry in international course accreditation processes for engineering and many other disciplines. The approach is based on the demonstrated student attainment of stated graduate attributes with the focus on outcomes rather than process. This approach encourages diversity and innovation in delivery and has brought significant benefits to engineering education (Palmer and Ferguson, 2009).

The outcomes-based approach generally requires a linking of course and unit level learning objectives with graduate outcomes. The use of taxonomies such as Bloom (1956) are commonly used in computer science education to describe learning outcomes and links to assessment (Lister, 2000) however the results of this survey and forum suggest there may be a lack of a shared understanding of the learning objectives for WIL experiences.

Learning objectives for WIL developed jointly by universities and industry (professional bodies and industry associations) will provide the basis for the development of a range of models of WIL each providing varying levels of authenticity (Figure 1). Models should include external forms where students go out to industry (e.g. industry placements, internships, field visits and community projects) as well as internal models where industry comes to students (e.g. guest speakers, case studies, industry linked projects and simulated experiences). The authenticity of each form of WIL should be evaluated according to the achievement of the

agreed learning objectives rather than just relying on personal opinion which in some cases have only recognised formal industry placements as the only valid form of WIL. Industry acceptance of the value of innovative learning experiences to provide students with the necessary understanding of aspects of professional practice will benefit all stakeholders.

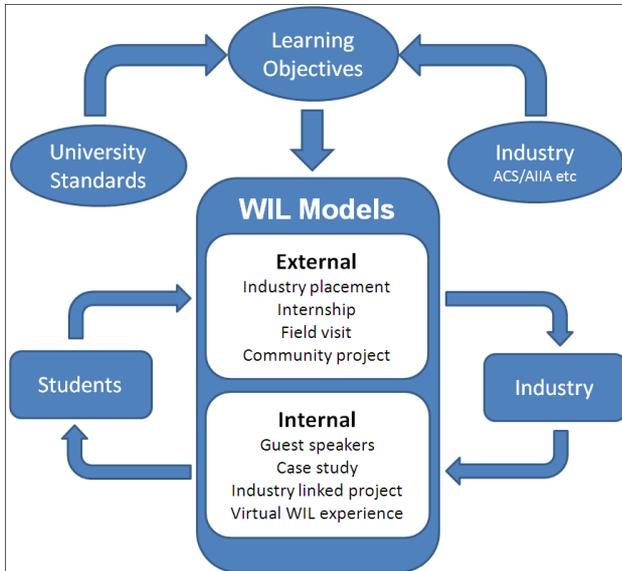


Figure 1: Learning Objectives and Models of WIL

This approach is similar to the current Engineering Australia accreditation requirements for professional engineering degrees (EA, 2008) which require “a minimum of 12 weeks of experience in an engineering-practice environment (or a satisfactory alternative)”. The EA requirements do state that there is “no real substitute for first-hand experience in an engineering-practice environment, outside the educational institution” however the requirements also state that “however it is recognised that this may not always be possible”, i.e., engineering students do not have to undertake 12 weeks of actual industry experience in an external organization in order to complete their degree but can achieve this requirement through alternative means. Valid learning experiences for professional practice include traditional placements as well as the “use of guest presenters, industry visits and inspections, an industry based final year project”. The EA requirements go on to indicate that: “The requirement for accreditation is that programs incorporate a mix of the above elements, and others - perhaps offering a variety of opportunities to different students - to a total that can reasonably be seen as equivalent to at least 12 weeks of full time exposure to professional practice in terms of the learning outcomes provided.”

This liberal interpretation of professional practice permits universities to provide a set of university-based learning experiences to achieve the ‘12 week experience’ requirement of EA accreditation. The professional practice requirement could be spread out over the duration of the degree program including providing a context to engage first year students as well as a professional preparation for final year students. The key requirement is that the experiences are authentic and can

be documented to demonstrate targeted graduate capabilities set for the program.

Using the Engineering Australia experience as an example, consideration should be given in the current revision of the Australian Computer Society Course Accreditation Guidelines to incorporate detailed guidelines for professional practice including stated learning objectives that have been endorsed by industry. Whilst the 12 week requirement appears to work well in the context of engineering, it is accepted that a similar requirement might not map well to the ICT disciplines. However, there would be wide ranging benefits in implementing a similar professional practice requirement for ICT degrees where the requirement is visible, significant and readily understood by the prospective and current students, teachers, parents, industry, government and the community in general, and provides scope for universities to innovate in the design of learning experiences and approaches.

5 Conclusion

The survey and forum discussions have revealed strong academic support for students gaining professional practice through a variety of WIL options. While placements may be a desirable component of WIL, circumstances may dictate alternative practices that may not be recognised by industry as authentic.

The development of a shared set of learning outcomes for WIL between academe and industry may provide recognition of the authenticity of innovative internal models of WIL such as virtual and simulated experiences as well as Final Year Projects related to industry needs. Learning outcomes that include experience of industry practices should be mutually acceptable whatever the processes used to produce the desired outcomes.

Future work in this area may include the identification of learning outcomes for WIL in conjunction with industry and the incorporation of learning outcomes for WIL and/or professional practice into ACS accreditation guidelines.

6 Acknowledgements

The following project team members contributed to the survey design: Jocelyn Armarego (Murdoch), Paul Bailes (UQ), Tanya McGill (Murdoch), Fazel Naghdy (UOW), Philip Ogunbona (UOW) and Chris Pilgrim (Swinburne).

7 References

- ACS (2009), ANZ ICT Accreditation Board, Accreditation Manual, Document 2A: Application Guidelines – Professional Level Courses, February 2009.
- AMP.NATSEM (2007). Generation whY?, AMP. NATSEM Income and Wealth Report, Issue 17. July.
- Bates A., Bates B., and Bates C., 2007. Preparing students for the professional workplace: who has responsibility for what? *Asia-Pacific Journal of Cooperative Education*, 2007, 8(2), pp. 121-129.
- BCA (2011) Lifting the Quality of Teaching and Learning in Higher Education, Business Council of Australia, from: <http://www.bca.com.au/DisplayFile.aspx?FileID=725>

- Billett S. (2011) Curriculum and pedagogic bases for effectively integrating practice-based experiences, ALTC Project Final Report, from: <http://www.altcexchange.edu.au/group/integrating-practiceexperiences-within-higher-education>
- Bloom B. S., (1956). *Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain*. New York: David McKay Co Inc.
- Bogdan, R.C. and Biklen, S.K. (2002). *Qualitative Research for Education: An Introduction to Theories and Methods*, 4th ed, Allyn & Bacon, London, UK.
- Boyatzis, R.E. (1998). *Transforming Qualitative Information: Thematic Analysis and Code Development*, Sage Publications, Thousand Oaks, CA.
- Boud, D. & Symes, C. (2000). Learning for real: work-based education in universities. In C. Symes & J. McIntyre (Eds), *Working Knowledge: the new vocationalism and higher education*, pp.14-29. Buckingham: Open University Press.
- EA (2008), *Accreditation Criteria Guidelines*, Document G02, Engineers Australia, August 2008.
- Forth, J. and Mason, G. (2003). *The Determinants and Effects of High-Level ICT Skill Shortages: Evidence from the Technical Graduates Employers Survey*. National Institute of Economic and Social Research, London. from: <http://www.niesr.ac.uk/research/ICT/TGES-7.pdf>.
- Hagan, D. (2004). 'Employer satisfaction with ICT graduates'. Sixth Australasian Computing Education Conference (ACE2004), Dunedin, New Zealand. *Conferences in Research and Practice in Information Technology*, vol. 30, pp. 119–23.
- Hare, J. (2011) Business takes dim view of academe, *The Australian Higher Education Supplement*, March 30th 2011.
- Kennan, M.A., Cecez-Kecmanovic, D., Willard, P. and Wilson, C.S. (2008). 'IS knowledge and skills sought by employers: a content analysis of Australian IS early career online job advertisements'. *Australasian Journal of Information Systems*, 15, pp. 57–78.
- Koppi, T. and Naghdy, F. (2009), *Managing Educational Change in the ICT Discipline at the Tertiary Education Level*, from: <http://www.altc.edu.au/system/files/resources/DS6-600%20Managing%20educational%20change%20in%20the%20ICT%20discipline%20March%202009.pdf>
- Lister, R. (2000). On Blooming First Year Programming, and its Blooming Assessment. *Proceedings of the Australasian Conference on Computing Education*, ACM Press, New York, NY, pp. 158-162.
- O'Connor, B. (2008). Work Integrated Learning (WIL): Transforming Futures Practice... Pedagogy... Partnership, Address to: World Association for Cooperative Education (WACE) Asia Pacific Conference, 1 Oct, 2008, from: http://www.deewr.gov.au/Ministers/OConnor/Media/Speeches/Pages/Article_081003_124044.aspx
- Ogunbona, P (2009), ALTC Project PP9-1274, "Addressing ICT curriculum recommendations from surveys of academics, workplace graduates and employers", from: <http://www.altc.edu.au/project-addressing-ict-curriculum-recommendations-uow-2009>.
- Palmer, S. and Ferguson, C. (2008). Improving outcomes-based engineering education in Australia, *Australasian Journal of Engineering Education*, Vol 14 No 2.
- Patrick, C., Peach, D., Pocknee, C., Webb, F., Fletcher, M., Pretto, G., (2008). The WIL [Work Integrated Learning] report: A national scoping study [Australian Learning and Teaching Council (ALTC) Final report].
- Pauling, J.W. and Komisarczuk, P. (2007). 'Review of work experience in a Bachelor of Information Technology'. Ninth Australasian Computing Education Conference (ACE2007), Ballarat, Victoria, January 2007. *Conferences in Research in Practice in Information Technology*, vol. 66, pp. 125–132.
- Poppins, P. and Singh, M. (2005). 'Work integrated learning in information technology education'. In *Information and Communication Technologies and Real-Life Learning: New education for the knowledge society*, T. van Weert and A. Tatnall (eds). Springer, New York, USA. Pp. 223–30.
- Mather, J (2010), Sector split over on-the-job year for IT students, *Australian Financial Review*, May 30th 2010.
- Shoikova, E. & Dwishev, V. (2004). University - Industry network, *Proceedings of the 27th Int'l Spring Seminar on Electronics Technology IEEE*, pp 510-514
- Smith, R., Mackay, D., Holt D. & Challis, D. (2008). Expanding the realm of best practices in cooperative industry-based learning in information systems and information technology: an inter-institutional investigation in Australian higher education, *Asia-Pacific Journal of Cooperative Education*, 9(1), 73-80

Trends in Introductory Programming Courses in Australian Universities – Languages, Environments and Pedagogy

Raina Mason¹ Graham Cooper¹ Michael de Raadt²

¹ Southern Cross Business School
Southern Cross University
Hogbin Drive, Coffs Harbour, New South Wales 2450
raina.mason@scu.edu.au
graham.cooper@scu.edu.au

² Moodle Pty Ltd
1/224 Lord St, Perth, WA 6000
michaeld@moodle.com

Abstract

This paper reports the results of a study of 44 introductory programming courses in 28 Australian universities, conducted in the latter months of 2010. Results of this study are compared with two censuses previously conducted during 2001 and 2003, to identify trends in student numbers, programming language and environment/tool use and the reasons for choice of these, paradigms taught, instructor experience, text used and time spent on problem solving strategies in lectures and tutorials. Measures of mental effort experienced during the solution of novice programming problems were also examined.

Keywords: introductory programming, programming languages, programming environments, Australian university courses, mental effort measures, census, trends.

1 Introduction

Programming skills are an essential part of Information Technology (IT) and Computer Science (CS) courses. Programming is generally regarded to be both complex and difficult, and introductory programming courses can suffer from high attrition rates and low levels of competency (McCracken et al. 2001). There has been much debate in the academic community about what languages, environments and paradigms should be used for students' first exposure to programming (Bruce, 2005; Cooper, Dann, et al., 2003; Kelleher and Pausch, 2005).

In 2001 a census of introductory programming courses at Australian universities was conducted (de Raadt, Watson & Toleman, 2002), which reported on the languages and environments/tools being used, the reasons for the choice of language, student numbers and the paradigm being taught. The census covered 57 courses at 37 of the 39 Australian universities. The other two universities did not offer programming courses. The

census was repeated in 2003 (de Raadt, Watson & Toleman, 2004) and expanded to include New Zealand universities. The 2003 census examined trends in languages and reasons for language choice, paradigms taught, tools and environments used, as well as new questions on texts employed, method of delivery to on-campus students, instructor experience and information about the teaching of problem solving strategies.

In the latter months of 2010 the census was to be repeated with all Australian universities. Participation was not as high as had been hoped, with a participation rate of 28 universities from the 39 that offered programming courses. A total of 44 out of 73 available programming courses were covered. While no longer a census, the study contains a large sample of the available first programming courses offered. The results of this 2010 study are reported in this paper, and have been compared to the results from the 2001 and 2003 censuses in order to identify longitudinal trends in language, tools and paradigms and to identify reasons for any such changes over the 10 year period. The basis for constructing interview questions and for conducting the study are described in the next section, followed by a discussion of the results and implications to teaching introductory programming.

2 Methodology

The previously collated list of participants from the 2001 and 2003 studies was used as a starting point for building a list of contacts for the current study. Information from university websites was also used to identify potential university programs and to collect contact numbers of administrative staff or academic staff responsible for those programs.

Once identified, each academic responsible for a particular introductory programming unit was sent an introductory email outlining the past two censuses. Participants were then contacted by phone within seven days to invite participation in the new census, and to arrange a convenient time for a phone interview of around 10 to 15 minutes duration.

All phone interviews were audio-recorded with the participant's permission. Notes on responses and comments were also entered manually into paper-based census forms as a backup to the recordings. Audio recordings were later transcribed and the data analysed.

Copyright © 2012, Australian Computer Society, Inc. This paper appeared at the Fourteenth Australasian Computing Education Conference (ACE2012), Melbourne, Australia, January 2012. Conferences in Research and Practice in Information Technology, Vol. 123. Michael de Raadt and Angela Carbone, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

It was found that the terminology used for a unit of study that is completed by students towards a degree varies between institutions, for example “subject”, “course” or “unit” are used. The interviewer used the terminology particular to each institution when conducting an interview to reduce possible confusion from ambiguity. In the remainder of this paper, the results are reported using the description “course” for the basic unit of study (usually studied over the period of a semester or session, in conjunction with other units of study), to be consistent with the last two censuses.

2.1 Questions

Questions repeated from the 2001 and 2003 censuses probed language and paradigm choice, teaching duration, instructor experience, textbooks, problem solving strategies and development tools.

The orbit of enquiry of the 2003 census was expanded to include questions regarding the mental effort (Paas and van Merriënboer 1993) required to understand and learn aspects of programming using the language(s) used in each course.

On this basis, the following question was asked:

- “How difficult do you think this language is for students to learn?” (9 point Likert scale, 1 = extremely easy, 9 = extremely difficult)

If an interviewee indicated that he or she used an environment or tool beyond a simple editor and command line compiler, then the following questions were asked:

- “Why was this environment/tool chosen?”
- “Is this environment/tool used for an initial part of the first programming course only, or throughout the first programming course?”
- “Is the environment/tool used in any other courses in the degree? If so, how many? Is it used in a different way in subsequent courses?”
- “How difficult do *you* find the environment to use (on a scale of 1 to 9 where 1 is extremely easy, 9 is extremely difficult)”
- “On average how difficult do you believe *the students* find the environment to use?” (9 point Likert scale)

All participants were also asked questions about the mental effort expended when solving a novice programming problem, on each of three measures:

- understanding and processing the problem statement;
- navigating or using the environment, tools or language; and
- learning from the problem and reinforcing previous concepts.

Participants were asked to estimate the mental effort expended on each of these three measures by themselves, by an average student, and by a student in the bottom 10% of their course.

3 Results and discussion

The results of this study are reported below, with comparison to the previous two censuses. For more accurate comparison, only the data from Australian universities in the 2003 census has been used.

3.1 Universities and courses

The 2010 study covered 44 of the 73 programming courses offered by Australian universities. A total of 28 of the 39 universities offering programming courses participated in the study (see Table 1).

	2001	2003	2010
Universities	39	40	40
Universities teaching programming	37	39	39
Introductory Programming courses	57	71	(44) 73
Total students in study (approx.)	19900	16300	7743
Average students per course	349	229	176

Table 1: University/Course Summary

3.2 Participation rate

Three instructors declined to participate. Several instructors were employed casually and were not available on campus at any time apart from their face-to-face teaching commitments, two instructors had retired recently and were not available for comment, and some instructors (and administrative staff) were unavailable during the 3 month period of conducting interviews for this study. All but one of the participants agreed to the audio-recording of the interview. This participant agreed to be interviewed with hand-written notes being taken on paper-based census forms.

3.3 Number of courses

Although the total number of programming courses offered has changed little since the 2003 census, instructors stated that often business-based IT programming courses, computing science programming courses and engineering programming courses had been amalgamated into one course. The reasons offered by participants in several institutions were that management had strategically decided to merge different courses and associated student cohorts to gain efficiencies due to economies of scale. These actions had been in response to declining numbers of students. At the same time, programming courses have appeared in non-traditional programming areas such as visual arts, keeping the number of introductory programming courses relatively static.

3.4 Student numbers

The declining number of students studying programming is a serious problem. Average numbers of enrolments per course have halved, falling from 349 in 2001 to just 176 in 2010. This follows a general trend in declining student enrolments in all areas and levels of ICT education - vocational, undergraduate and postgraduate - with 17 436 total domestic ICT enrolments in 2001 falling to just 7 470 domestic students in 2008 (ACS, 2010).

This is despite strong growth prospects for employment in the IT industry, with 7 out of 11 ICT job designations in the “Job Prospects Matrix” (DEEWR, 2011) considered to have ‘strong’ to ‘very strong’ growth prospects for future employment. The widening gap between increasing demand and declining domestic student numbers may give rise to a significant short-fall

of suitably educated and skilled IT professionals in the near future.

3.5 Languages

3.5.1 Choice of language(s)

There were 20 different languages being taught by the academic staff interviewed, which is a greater diversity of languages than displayed in each of the last two studies. During 2001, only nine languages were recorded, and in 2003 this number had reduced to eight languages. In 2010, the number of languages taught over the duration of a course ranged from 1 to 6 (see Table 2), with the vast majority teaching just one language.

No. of languages	1	2	3 to 6
Courses	37	4	3

Table 2: number of languages in a course

Some languages were only used for a short time during a course, with another language being used for most of the course. For example, in one case Alice was used for the first two weeks, followed by Java. If only these ‘primary’ languages are counted, 12 languages were used by instructors. One of these languages is "MaSH", a language created as a staged subset approach to Java (Rock, 2011). For the purposes of this study, this language has been counted as a variant of Java. The number and percentage of courses using each primary language, as well as the weighted percentage by students, are shown in Table 3.

Language	Courses	%age	Weighted by students
Java	16	36.4%	38.4%
Python	6	13.6%	19.2%
C	5	11.4%	11.7%
C#	4	9.1%	8.0%
Visual Basic	4	9.1%	5.1%
C++	3	6.8%	4.8%
Processing	2	4.5%	5.2%
Alice	1	2.3%	0.9%
Fortran	1	2.3%	3.9%
Javascript	1	2.3%	1.5%
Matlab	1	2.3%	1.3%

Table 3: 2010 Languages

The set of top three languages has changed since the 2003 census. Java continues to hold the place of the most popular language, a result which is in accordance with a recent survey of programming courses in the USA (Davies et al. 2011). C++ and Visual Basic have decreased in popularity and fallen out of the top three. Python, under development since 1990, is the second most popular language in 2010, followed by C. According to the TIOBE programming language index (TIOBE Software 2011), industry use of Python had the greatest growth in popularity of any language in the years of 2007 and 2010.

The relatively new language C# is the fourth most popular language, seemingly replacing Visual Basic, which has dropped to just 5.1% of student capture. This apparent replacement is not surprising, as both Visual Basic and C# are Microsoft products and C# is considered to be more modern and popular than Visual Basic (TIOBE Software 2011).

A comparison of the language use across the 2001, 2003 and 2010 studies is provided in descending order by percentage of courses (Table 4) and by student numbers (Table 5).

Language	2001	2003	2010
Java	40.4%	40.8%	36.4%
Python	0.0%	0.0%	13.6%
C	7.0%	12.7%	11.4%
C#	0.0%	0.0%	9.1%
VB	24.6%	26.8%	9.1%
C++	14.0%	11.3%	7.0%
Processing	0.0%	0.0%	4.5%
Fortran	0.0%	1.4%	2.3%
Javascript	0.0%	0.0%	2.3%
Matlab	0.0%	1.4%	2.3%
Alice	0.0%	0.0%	2.3%
Haskell	5.3%	4.2%	0.0%
Eiffel	3.5%	1.4%	0.0%
Delphi	1.8%	0.0%	0.0%
Ada	1.8%	0.0%	0.0%
jBase	1.8%	0.0%	0.0%

Table 4: language comparison by courses

Language	2001	2003	2010
Java	43.9%	44.4%	39.0%
Python	0.0%	0.0%	19.5%
C	5.5%	10.6%	11.9%
C#	0.0%	0.0%	8.2%
Processing	0.0%	0.0%	5.3%
VB	18.9%	16.4%	5.2%
C++	15.2%	18.7%	4.9%
Fortran	0.0%	0.7%	3.9%
Javascript	0.0%	0.0%	1.5%
Matlab	0.0%	1.0%	1.3%
Alice	0.0%	0.0%	0.9%
Haskell	8.8%	6.0%	0.0%
Eiffel	3.3%	2.1%	0.0%
Delphi	2.0%	0.0%	0.0%
Ada	1.7%	0.0%	0.0%
jBase	0.8%	0.0%	0.0%

Table 5: Language comparison by students

Figures 1 and 2 chart the changes in popularity of the top 4 languages in each of these three studies (Java, VB, C++

and Haskell in 2001, Java, VB, C++ and C in 2003, and Java, Python, C and C# in 2010), by percentage of courses offering the language, and then weighted by students.

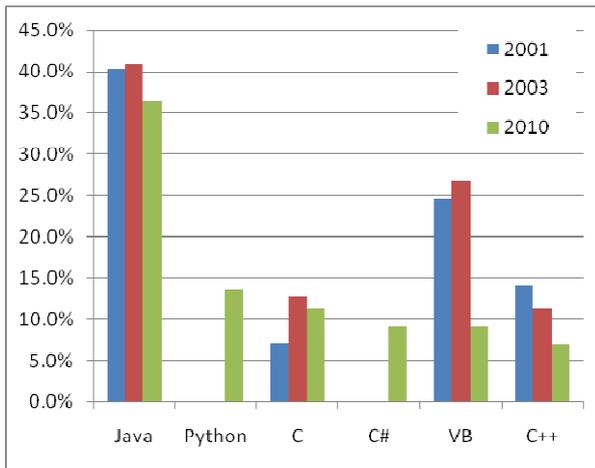


Figure 1: Top 4 languages of each study (by %age of courses)

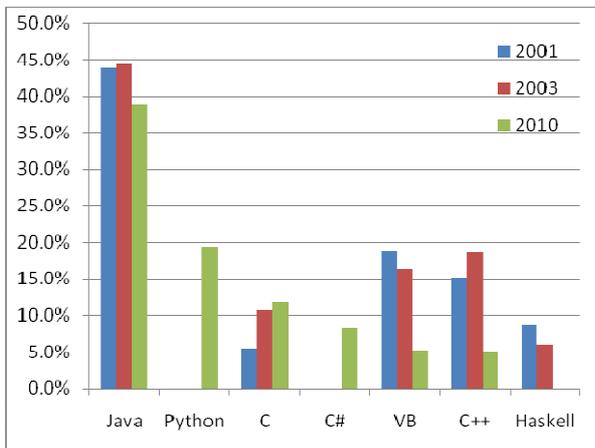


Figure 2: Top 4 languages of each study (weighted by student numbers)

3.5.2 Reason for choice of language

Instructors were asked in the 2001 census and the 2010 study about the reasons for their choice of language. More than one reason could be offered.

In 2001 the most commonly provided reason for choosing a language was industry relevance and/or marketability to students (with 56.1% of participants identifying this as a reason). The second most commonly provided reason for choosing a language was “pedagogical benefits”, with one third (33.3%) of the instructors presenting this as a reason for language choice. The results of the 2010 study presented a substantial shift in the frequency of these reasons being given for language choice, with industry relevance and marketability declining (to 48.8%) and pedagogical benefit rising (to 53.5%).

The 2010 survey also identified the emergence of several new reasons. Instructors mentioned “the availability of a community and online help”, “extendability and libraries available”, “platform independence” (as opposed to “limitations of

OS/machines” as in the 2001 census), “ease of installation”, and “interpreted language”, with no need to compile. Some of these reasons reflect the rise of the open source community over the time period, as well as perhaps a greater choice of operating systems by students. Table 6 shows the change in percentages of instructors’ reasons for language choice between 2001 and 2010, ordered by frequency in 2001.

Reason	2001	2010
Used in industry / marketable	56.1%	48.8%
Pedagogical benefits of language	33.3%	53.5%
Structure of degree/dept politics	26.3%	32.6%
OO language	26.3%	16.3%
GUI interface	10.5%	7.0%
Availability/Cost to students	8.8%	4.7%
Easy to find appropriate texts	3.5%	2.3%
OS/Machine limitations of dept	1.8%	4.7%
Online community and help available	0%	9.3%
Platform independence	0%	9.3%
Extensions/Libraries available	0%	7.0%
Interpreted language	0%	4.7%
Ease of installation	0%	2.3%

Table 6: Reasons for language choice

Figure 3 shows the comparative frequencies of reasons given in the 2001 census with the 2010 results, ordered by the 2010 results.

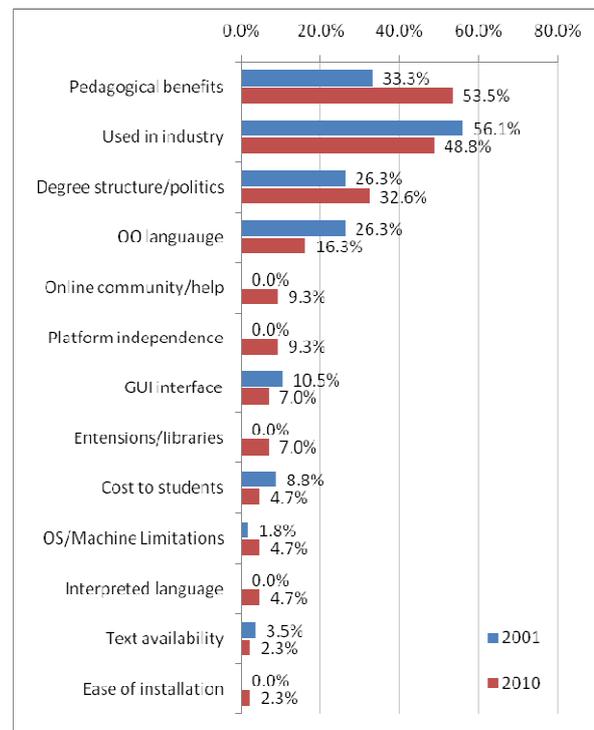


Figure 3: Trends in reasons for language choice

This figure clearly shows that although “industry relevance and marketability to students” is still seen as important, the reasons for choosing a language have

shifted to be more strongly inclusive of pedagogical factors associated with both ease of learning and availability of support.

The intersection of industry relevance and pedagogy can be seen in the reasons given by those instructors who chose Python as their teaching language. As previously noted, Python is one of the more popular languages according to the TIOBE Programming Community Index (TIOBE Software 2011), which indicates the popularity of programming languages in industry (numbers of skilled engineers world-wide and third-party vendors) and in training courses. However *all* of the instructors who chose Python commented that the reason they chose the language was because it was perceived as easier for students to learn. Only one instructor gave ‘industry relevance’ as a secondary reason for the choice of Python, and this was due to its association with the Google Apps engine. The comment was also made that “we have to cater for prep [sic] students and keep IT students happy, so we have to find a balance between these”.

The reason “structure of degree/department politics” was the only other reason given in 2001 that increased in frequency in the 2010 study. As previously mentioned, in several cases two or more introductory programming courses in various disciplines such as engineering, business and computer science had been merged into one course. The language that was chosen for this one course became either a legacy language from one of the previous courses, or a language chosen to try to cater to a broader profile of students pushed into a narrower stream of programming teaching. Several instructors expressed frustration at the need to cater to a range of students with differing backgrounds, experience and capabilities. Typical comments included “We see students with a range of skills - from no experience to some with some computing in high school” and “IT students are not the same as CS students”.

3.6 Paradigm taught

In common with the 2001 and 2003 censuses, the 2010 study showed that most instructors choose to teach using a procedural paradigm. Some instructors - 8 from the 44 interviewed - also reported that they taught primarily procedurally but introduced some object-oriented concepts. For example, they may mention objects or use the other terminology of object-oriented programming to prepare students for further courses that covered OO programming. Some used objects but “in a procedural way”. For the purposes of comparison and consistency with previous studies, these have been designated under the procedural paradigm.

Some instructors described how they used either a mix of paradigms - for example they started with procedural and then used the last 6 weeks to cover OO concepts - or suggested that they taught more than one language and taught each language in different ways. The number of courses teaching in each paradigm is given in Table 7.

Longitudinal trends in paradigms taught are shown in Figure 4. The downwards trend in the numbers of instructors teaching objects-first is clearly shown and the use of the functional paradigm has nearly disappeared.

Paradigm	Courses	%age
Procedural	24	54.5%
Object-Oriented	11	25.0%
Mixture	8	18.2%
Functional	1	2.3%

Table 7: Paradigms taught

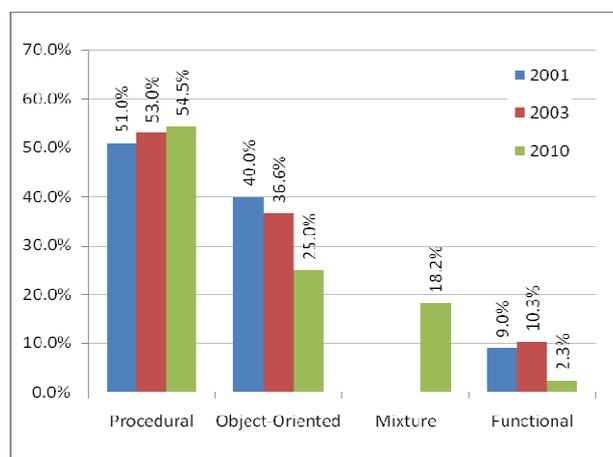


Figure 4: Trends in paradigms taught

3.7 On-campus hours

Instructors were asked about the time that on-campus students spent in lectures, tutorials and practicals each week. Although the average hours spent in each did not differ greatly from 2003 (see Table 8), several institutions stated that they either have no lectures at all, instead presenting in a more interactive ‘workshop’ format, or that all lectures were available only online via online classroom software and/or audio or video recordings. Most instructors commented that all course materials were available online and that often students did not attend classes, instead choosing to download materials from home, even when the course was not offered by distance education.

Several interviewees commented on the success of the ‘workshop’ approach: “Workshops are a much more successful way of teaching programming. Give them small bits and then have them do it straight away.”

	Lecture	Tutorial	Practical	TOTAL
2003	2.2	0.6	1.8	4.6
2010	2.1	1.1	1.2	4.4

Table 8: Hours in class on-campus

3.8 Instructor Experience

Instructors were asked how many years they had taught introductory programming. The average amount of experience has risen since 2010 (see Table 9). However it should be noted that casual teachers were often not available for interviews, and this may have skewed the results.

	Minimum	Average	Std Dev	Maximum
2003	0.5	8.6	7.2	30
2010	2	12.3	7.3	30

Table 9: Instructor experience in years

3.9 Texts used

Of the 44 courses in this current study, 11 used no text at all, and many instructors commented that they encouraged students to use online resources or to find their own text. Most of the remainder used language, or environment-specific texts, with no real commonalities. Four more generic texts - “Connecting with Computer Science” (Anderson, Hilton and Ferro 2010), “Simple Program Design” (Robertson 2000), “Programming Language Concepts” (Sebesta 2007) and “A simple and generic introduction to OO Algorithm Design” (Robey undated) - were used, each in only one course.

3.10 Problem Solving Strategies

Instructors were asked how much time they dedicated during class time to teaching and discussing problem solving strategies within the context of programming.

The average percentage time given to problem solving, along with standard deviations, have remained stable from 2003 to 2010, and are presented in Table 10. The percentage of times given to problem solving in both lectures and tutorials remain unchanged in this period.

	Lecture		Tutorial	
	Average	Std Dev	Average	Std Dev
2003	29%	22%	46%	36%
2010	29%	26%	44%	33%

Table 10: percentage of class time dedicated to problem solving

It is worth noting that while the means between years are very close for both lectures and tutorials, the standard deviations on these measures are relatively large, indicating substantial variations between different courses in how much time they dedicate explicitly to problem solving.

Some participants indicated that ‘problem solving’ had been moved into a separate course and so did not deal with these strategies in the programming course. In contrast, others stated that problem solving was implicit in everything they did in the lectures and tutorials, although they did not explicitly teach problem solving strategies.

Although the average percentage of time given to focusing upon problem solving has remained static from 2003 to 2010, there may be differences in how problem solving is embedded within curriculum structures, classroom delivery and learning activities between these two dates. This information, however, lies beyond the bounds of the current paper.

3.11 IDEs and tools

3.11.1 Choice of IDE/tools

Some languages (including Alice and Processing) require the use of a specific environment. Instructors who did have a choice of environments or tools chose a wide variety or none at all. Microsoft Visual Studio was the most popular IDE. The use of the teaching environment BlueJ continued to increase, from 4% in 2001 to 17.5% in 2010. Within the ‘other’ category, note should be made of the “IDLE” IDE (for use with Python), at 12.5% and Eclipse, used by 7.5% of instructors.

The largest change has been the movement away from using text editors and command line compilers only - from approximately 45% of these instructors using no IDE/tool in 2001 and 2003, to just 20% in 2010. Figure 5 shows the trends in environment use over the course of the three studies.

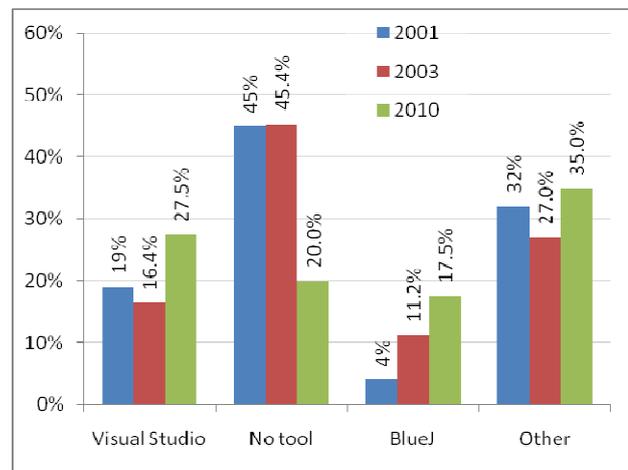


Figure 5: Trends in environment use

Two instructors reported using a specialised learning environment (such as Alice or BlueJ) for an initial part of the course, followed by a more industry-standard IDE such as Visual Studio or Eclipse. Others used the learning environment throughout the first programming course.

3.11.2 Reasons for choice of environment

At the time of the 2003 census, most instructors were choosing to *not* use an IDE, if they were not forced to do so by their choice of language. Anecdotally this was due to the perceived overhead of instructing students on that tool, hence instructors preferred to use a simple editor and command line compiler. During the 2010 study, instructors were directly asked about the reason or reasons that they had chosen a particular IDE or tool. The top 10 reasons for choosing an environment/tool (excluding those where the language is part of the environment, such as Alice and Processing) is given in Table 10.

Other reasons included “associated text is very good”, “cross-platform”, “plugins available”, “ease of installation” and “GUI”. Some instructors offered reasons as to why they did not choose to use an IDE, even though this was not explicitly asked as part of the study. These included “students need to become familiar with the command line” and “we don’t have time for that”.

Reason	Count	% courses
Pedagogical reasons	15	41.7%
Packaged with language	14	38.9%
Cost to students	13	36.1%
OS limitations of dept.	7	19.4%
Uncomplicated/Ease of use	6	16.7%
Industry relevance	4	11.1%
Supports OO paradigm	4	11.1%
Visual cues/Visual debugger	4	11.1%
Student motivation	3	8.3%
Open source	3	8.3%

Table 10: Top 10 reasons for environment choice

The most frequent reason given for choice of environment (provided by 41.7% of instructors) was pedagogically-based. This is consistent with the reasons provided by participants for choice of language, where pedagogical reasons was also identified as the most frequent reason for choice (of language).

Associated comments offered by instructors emphasised that some environments assisted learning by allowing students to concentrate on the concepts of programming, rather than the specifics and nuances of a language:

- “can think about objects instead of thinking about the language itself”;
- “[an] introduction without the stress of having to worry about syntax rules. It reinforces what happens when you use a loop so the frog jumps once or it just keeps jumping. You have to understand the concept, so its concept reinforcement.”;
- “We found in the past that students were having trouble with understanding what all the features of the main method are, and some of the initial concepts we had to abstract away – we had to say ‘treat this as magic, you have to do it’”;
- “...mostly to encourage students who have no programming background, and strengthens the concepts of loops, and iterations and all those things”.

Some instructors stressed that the environment chosen was simple whilst still including tools to reduce the complexity of compiling and building:

- “it’s really a smarter text editor with buttons for compiling and building. Nothing like Visual Studio or Eclipse. ...so we can concentrate on language syntax”;
- “To reduce the amount that students had to learn in order to get to the heart of programming”;
- “just its simplicity”;
- “simple, not confusing”.

Others pointed at an IDE’s perceived intrinsic superiority to simple editors/command line compilers, due to the inclusion of helpful tools:

- “why get them to travel by horse when they can travel by car?”;

- “the tools that it provides for students are wonderful compared to using just a basic text editor - they offer nothing”.

There has been an apparent shift from viewing the use of an IDE as an additional overhead, to seeing it as reducing the amount a student has to learn, or as a tool to help the student to learn. Whether this viewpoint is correct is debatable, and may be a function of what is being taught (central concepts to programming in general, or a language specific syntax), the student’s ability and experience, and aspects of the language or environment itself.

3.12 Mental effort

3.12.1 Novice programming, mental effort and cognitive load

Mental Effort refers to the level of conscious focus of attention one has to give to a task, whether it is cognitive, physical, or a bit of both (Paas et al. 2003).

International educational and scientific computing body ‘The Association for Computing Machinery (ACM)’ (2008) suggests that learning the three generic concepts of sequence, iteration and selection is an integral part of all first programming courses. A novice programmer will need to acquire this knowledge base as schemas (Chase & Simon 1973) and automate them, whereby they can be applied with relatively low levels of conscious attention (Cooper & Sweller 1987).

Novice programming problems that require the student to learn and use any of these three concepts can be said to have three sources of mental effort:

- understanding the problem and what is required, and deciding on the best structures to use to solve the problem;
- navigating and using the environment, tools and language, in an attempt to solve the problem; and
- learning how to best use these structures so they can be used in further, more difficult problems.

These three aspects equate to three identified sources of cognitive load on working memory while learning: intrinsic, extraneous and germane (Sweller, van Merriënboer & Paas 1998).

Intrinsic cognitive load refers to the innate relative difficulty of a body of to-be-learned information. It is effectively set, defined by the content.

Extraneous cognitive load refers to the load generated by the format of instructional materials and/or to the performance of learning activities. Some formats and/or activities hinder learning by loading the learner with unnecessary information and/or tasks. This source of cognitive load is variable, determined by the learning materials.

Germane cognitive load refers to load devoted to the processing, construction and automation of schemas - knowledge structures in long-term memory. This is not simply a measure of motivation, but refers to the dedicated commitment of cognitive resources to the successful process of cognitive acquisition of new to-be-learned information.

The general strategy sought in many instructional settings is to reduce extraneous load, and to direct the

subsequently released cognitive resources towards the germane efforts associated with schema acquisition and automation. The term ‘mental effort’, defined above, is used in the current study as a means of evaluating the *cognitive load* (Sweller 1998) associated with various aspects of learning programming.

3.12.2 Measures of mental effort

Instructors were asked about the three sources of mental effort expended whilst solving a novice programming problem:

- understanding and processing the problem statement,
- navigating or using the environment, tools or language, and
- learning from the problem and reinforcing previous concepts.

Instructors were asked to rate their own levels of mental effort on each of these three factors using a 9 point Likert scale, where 1 = "no mental effort" and 9 = "extreme mental effort". Instructors were subsequently asked to rate their expectations regarding these three factors of mental effort for an average student in their introductory programming course, and then again, for a student in the “bottom 10%” of the course performance.

Table 11 shows the mean, median and mode for each of these cognitive load areas, for instructors, the average student and students in the ‘bottom 10%’.

		instructor	average student	bottom 10% student
intrinsic	mode	2	6	9
	median	2	6	8.5
	mean	2.8	6.0	7.8
	std dev	1.8	1.6	2.1
extraneous	mode	2	5	9
	median	2	5	8
	mean	2.4	4.9	7.7
	std dev	1.1	1.5	1.3
germane	mode	2	7	9
	median	2	5.5	8.5
	mean	3.2	5.6	7.6
	std dev	2.1	1.9	2.3

Table 11: Levels of mental effort

Note that there were some participants who did not quantify a response for an aspect of this series of questions, particularly for the ‘bottom 10%’. Instead they offered comments and discussion. These are removed from this first series of analyses, and are beyond the scope of the current paper, but will be explored in a further paper.

3.12.3 Comparisons of mental effort

For each of these three areas of cognition (understanding the problem statement, using the environment, and reinforcing previous concepts) a series of Wilcoxon Signed Rank tests were performed, firstly comparing the

self-rating of the instructor to that anticipated to be experienced by an ‘average student’, and then comparing the anticipated level to be experienced by an average student to one who is in the “bottom 10% of students”.

Table 12 shows the results, using Wilcoxon Signed-rank test (one-tailed) for all measures:

In summary, these results indicate that for *each* of these three sources of cognitive load, the instructors in the introductory programming courses rated their own levels of required mental effort to be low, and that they expected that average students would need to exert higher levels of mental effort than themselves - ‘above average’.

Additionally, for *each* of these three sources of cognitive load, the participants rated the anticipated mental effort to be experienced by a student in ‘the bottom 10%’ to be higher again, compared to an average student, in the rating of high to extreme mental effort.

Instructor -> average student (greater mental effort)					
	W	Ns/r	z	p	n
Understanding and processing the problem statement	811	43	4.39	<0.0001	43
Navigating/using the environment, tools or language	838	41	5.43	<0.0001	43
Learning from the problem/ reinforcing previous concepts	647	40	4.34	<0.0001	42
Average -> bottom 10% student (greater mental effort)					
	W	Ns/r	z	p	n
Understanding and processing the problem statement	207	24	2.95	0.0016	25
Navigating/using the environment, tools or language	276	23	4.19	<0.0001	25
Learning from the problem/ reinforcing previous concepts	144	20	2.68	0.0037	21

Table 12: Wilcoxon Signed-rank test

Average students need to “work harder” with their cognitive resources than the lecturer, and for less able students this is further exacerbated. This indicates that it is unlikely that these students will learn effectively, no matter how much effort they may put in.

Further research is needed to explore the reasons why some students struggle and the ways in which various environments and/or languages may aid or hinder learning these generic concepts in programming.

4 Further discussion

The shrinking number of students enrolled in introductory programming courses continues to be a concern. Average numbers of students per course have approximately halved in the 10 years since the first census was conducted. This is echoed by the ACS (2010) figures which show a drop in domestic ICT enrolments of 57% over the time period 2001 to 2008.

However ICT enrolments in Australian universities stayed relatively constant over the time period 2006 - 2008 (the latest figures available), primarily as a result of an influx of international student enrolments (ACS 2010). At the time of the latest figures in 2008, over 60% of the ICT enrolments at Australian universities were

international students, and the number of domestic ICT students were still decreasing (Table 13).

	Domestic		International		Total
2006	8198	44.8%	10087	55.2%	18285
2007	7839	43.0%	10384	57.0%	18223
2008	7470	38.6%	11896	61.4%	19366

Table 13: ICT enrolments 2006 - 2008

Languages taught in Australian universities continue to be dominated by Java, however there is a much wider diversity of languages than at the time of the last census. Instructors were also asked whether they had plans to change the first language, and although only four from 44 answered affirmatively, four others communicated that a change was being considered.

Languages that are seen as particularly beneficial for learning purposes (rather than for industry use) are becoming more popular, such as Python, Alice and Processing. This is supported by the stated reasons for choice of language or languages - even though "industry relevance" is still important, pedagogical factors were the reason for most instructors' language choice.

Of interest were the few courses that introduced novice users to three or more languages. The choice of several languages was either made because students needed to know these languages for further units, or as an attempt to show similarities in constructs and approaches in programming problem solving, despite differences in syntax and development environment. The mental effort results reported in this study suggest that for novices and in particular, for less able students, this latter approach is problematic at best, resulting in excess cognitive load and ineffective learning.

Online communities and resources have become more important. There were 9.3% of instructors gave "online community/help" as a factor in their choice of language, and a quarter of instructors set no text, many commenting that they encouraged students to use online resources.

Another change that has appeared since the last running of census is that students are anecdotally using a wider range of operating systems, and this has influenced instructors' choice of languages and environments. "Platform independence" was given as a reason for language choice by 9.3% of instructors, and was also mentioned as a factor in choice of IDE choice.

The focus on the object-oriented paradigm and the objects-first approach to learning programming appears to have reduced since the 2003 census. Those teaching objects-first dropped from 36.6% of courses in 2003 to just 25% in 2010. In addition, the reason "object-oriented language" for language choice was given by just 16.3% of instructors in 2010 compared to 26.3% in 2001. Interestingly, the use of BlueJ, an environment which supports the teaching of objects-first using Java, increased from 4% of courses to 17.5% over the same period.

The shifts towards greater emphasis upon pedagogy and pedagogical reasons for choices of languages and environments invites further exploration into some of the cognitive overheads experienced by students, particularly

less able students in the cohort. The mental effort measures and differences reported in this paper indicate that instructors are aware of the higher levels of mental effort experienced by average students over that expended by themselves solving the same problem, and that less able students are in many cases experiencing 'extreme' cognitive load while trying to solve novice programming problems.

Most instructors indicate that the 'bottom 10%' of students may represent more than one student profile - for example, those who are trying but failing to succeed, those who don't try, and those who are absent. These comments are explored further in a separate paper.

5 Further work

The numbers of students enrolled in introductory programs, as well as domestic ICT enrolments as a whole, continue to trend downwards, and more investigation is required to determine why this is happening.

Several participants in this study suggested that their courses had been formed by the amalgamation of more than one course that was originally part of computer science, information technology, engineering or business programs. An exploration of the current percentages of each cohort of students in these new courses, and their relative success would be of interest.

The authors of this paper are also exploring the implications of cognitive load, and its three primary sources, being intrinsic to the complexity of the content, extraneously related to the instructional materials and environments, which for programming includes the role of languages and environment, and germanely through the conscious, deliberate, focus of attention to the acquisition and automation of associated schemas.

Ironically, the students who are most in need of acquiring and automating schemas, those in the lower end of ability in a student cohort, are those that are least capable of doing so, due to excessive levels of cognitive load. Of primary interest is the potential for selection of language and environment to hold the potential of lowering a source of extraneous cognitive load, to thus free cognitive resources for application to germane usage, with the intent of facilitating learning.

6 Acknowledgements

The authors would like to thank the participants in this study for their involvement.

7 References

- Anderson, G., Hilton, R. & Ferro, D. (2010): Connecting with Computer Science. 2nd Ed. Cengage Learning.
- Association for Computing Machinery (2008): Computing Curricula - Information Technology 2008. Curriculum Guidelines for Undergraduate Degree Programs in Information Technology. http://www.acm.org/education/curricula/IT2008_Curriculum.pdf. Accessed 29 Aug 2011.
- Australian Computer Society (ACS) (2010): Australian ICT Statistical Compendium 2010, <http://www.acs.org.au/2010compendium> Accessed 18 Mar 2011.

- Bruce, K.B. (2005): Controversy on how to teach CS 1: A discussion on the SIGCSE-members mailing list. *Inroads - The SIGCSE Bulletin* **37**: 111-117.
- Chase, W.G. & Simon, H.A. (1973): Perception in chess. *Cognitive Psychology* **4** (1): 55–81.
- Cooper, G., & Sweller, J. (1987): Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of Educational Psychology* **79** (4): 347–362.
- Cooper, S., Dann, W. and Pausch, R. (2003): Teaching objects-first in introductory computer science. *Proc. ACM 34th SIGCSE technical symposium on Computer science education*, New York NY, USA, 191-195, ACM Press.
- Davies, S., Polack-Wahl, J.A. & Anewalt, K., 2011. A snapshot of current practices in teaching the introductory programming sequence. *SIGCSE '11 Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. ACM Press, pp. 625 - 630.
- De Raadt, M., Watson, R. and Toleman, M. (2002): Language trends in introductory programming courses. *Proc. Informing Science and IT Education Conference*, Cork, Ireland, Cohen, E. and Boyd, E. (Eds). InformingScience.org
- De Raadt, M., Watson, R. and Toleman, M. (2004): Introductory programming: what's happening today and will there be any students to teach tomorrow? *Proceedings of the sixth conference on Australasian Computing Education*. Dunedin, New Zealand, **30** , Australian Computing Society, Inc.
- DEEWR (2011): Australian Jobs 2011. <http://www.deewr.gov.au/Employment/ResearchStatistics/Pages/AustralianJobs.aspx>. Accessed 22 Aug 2011.
- Kelleher, C. and Pausch, R. (2005): Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, **37**(2):83-137. ACM Press.
- McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagam, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I. and Wilusz, T. (2001): A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*. **33**(4):125-180.
- Paas, F.G.W.C & van Merriënboer, J.J.G. (1993): The efficiency of instructional conditions: An approach to combine mental-effort and performance measures" *Human Factors* **35**(4): 737-743.
- Paas, F, Tuovinen, J. E., Tabbers, H. & Van Gerven, P.W.M. (2003): Cognitive Load Measurement as a Means to Advance Cognitive Load Theory. *Educational Psychologist*. **38**(1):63-71. Lawrence Erlbaum Associates, Inc.
- Robey, M. (undated): A simple and generic introduction to OO algorithm design. Self-published. <http://www.computing.edu.au/~mike/ST151Book.pdf> Accessed 24 Aug 2011.
- Rock, A. (2011): MaSH (Making Stuff Happen). <http://www.ict.griffith.edu.au/arock/MaSH/> Accessed 23 Aug 2011.
- Robertson, L.A. (2000): Simple Program Design. Nelson Australia.
- Sebesta, R. (2007): Programming Language Concepts. Addison Wesley.
- Sweller, J. (1988): Cognitive load during problem solving: Effects on learning. *Cognitive Science* **12** (2): 257–285.
- Sweller, J., Van Merriënboer, J., & Paas, F. (1998). Cognitive architecture and instructional design. *Educational Psychology Review* **10** (3): 251–296.
- TIOBE Software (2011): TIOBE Programming Community Index - Long Term Trends. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Accessed 22 Aug 2011.

Teaching Novice Programming Using Goals and Plans in a Visual Notation

Minjie Hu

Department of Information Science
University of Otago
PO Box 56, Dunedin 9054
minjiehu@infoscience.
otago.ac.nz

Tairawhiti Campus
Eastern Institute of Technology
PO Box 640, Gisborne 4010
New Zealand

mhu@eit.ac.nz

Michael Winikoff

Department of Information Science
University of Otago
PO Box 56, Dunedin 9054
New Zealand
mwinikoff@infoscience.
otago.ac.nz

Stephen Cranefield

Department of Information Science
University of Otago
PO Box 56, Dunedin 9054
New Zealand
scraneffield@infoscience.
otago.ac.nz

Abstract

Introductory programming courses have been continuously reported as having a high rate of failure or withdrawal. This research aims to develop a new approach for teaching novice programming, which is both easy to introduce and effective in improving novice learning. Our approach combines three key ideas: using a visual programming language; using strategies, specifically using the concepts of “goal” and “plan”; and having a well-defined process. We present a way of representing goals and plans in a visual notation together with a plan library that we developed in a visual programming environment (VPE). A key feature of the approach is that a design, i.e. an unmerged “plan network”, is executable and can be tested, giving feedback in the VPE. Furthermore, we describe a detailed process for using existing plans and building new plans in the VPE. This approach had been evaluated experimentally and the results indicated its potential to significantly improve teaching programming to novices.

Keywords: Goal, Plan, Visual Notation, Process of Programming.

1 Introduction

Although a wide range of approaches have been proposed to improve novices’ learning of programming (Kay *et al.* 2000, Pears *et al.* 2007, Robins, Rountree, and Rountree 2003, Rößling *et al.* 2008), there continues to be a high rate of failing or withdrawing from the first programming course (Lahtinen, Ala-Mutka, and Järvinen 2005, Sykes 2007). A number of reasons have been proposed for this, such as “fragile” knowledge of programming concepts (Lister *et al.* 2004, McCracken *et al.* 2001), lack of problem-solving strategies and plans (de Raadt 2008, Winslow 1996), and lack of detailed mental models (du Boulay 1989, Winslow 1996). There seems to be a broad consensus that “*novice programmers know the syntax and semantics of individual statements, but they do not know how to combine these features into valid programs*”

(Winslow 1996, page 17). This research therefore focuses on the heart of the matter: how to teach novices to construct programs.

The basis for this work is the hypothesis that what is needed is a *process* that students can follow, along with a structured means of representing the parts of a solution using an easy-to-use notation. Specifically, we conjecture that combining goals and plans with a detailed process will yield an effective means for teaching programming. Our proposed approach thus combines three ideas: using a Visual Programming Environment (VPE), using goals and plans, and having a well defined process. This combination is novel and carefully motivated. We have chosen to use a VPE (Kelleher and Pausch 2005) (specifically Scratch¹ (Resnick *et al.* 2009)) since VPEs aim to provide an attractive, easy, and fun way for novices to learn programming. In VPEs, such as Alice (Sykes 2007) and Scratch, programs are built by dragging and dropping statement blocks, which helps to prevent syntax errors and avoids the need to learn and memorize syntax. The idea of goals and plans is based on the finding that experts use strategies to solve programming problems (Soloway 1986). We follow other researchers (de Raadt 2008, Guzdial *et al.* 1998) in using goals and plans. We represent them explicitly, by devising a visual notation for goals and plans, and extending the Scratch language with an explicit representation for plans. Finally, we define a detailed process to guide novices through the activities of programming. This process is not just a high-level sequence of steps, but includes detailed guidance for how to perform sub-steps in this process.

The remainder of this paper is structured as follows. In Section 2, we discuss related work, in particular work that uses the goal and plan concepts and a programming process. Section 3 presents the explicit representation of goals and plans. In Section 4, a well defined programming process utilising an existing plan library is described, and in Section 5, a process for building new plans is given. Section 6 presents an evaluation of the proposed approach. Finally, we conclude in Section 7.

2 Literature Review

A *goal* is a certain objective that a program must achieve in order to solve a problem (Letovsky and Soloway

Copyright © 2012, Australian Computer Society, Inc. This paper appeared at the 14th Australasian Computing Education Conference (ACE 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 123. M. de Raadt and A. Carbone, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

¹ <http://scratch.mit.edu>

Literature Study	Goals	Plans /Patterns / Schemata	Detailed Process	Test /Debug	Language
Soloway and his colleagues (1980s & 90s)	Yes	Plans in Code	Weak	N/A	Pascal
Porter and Calder (2003)	Weak	Patterns in Code	Weak	No	C
de Barros et al. (2005)	Weak	Patterns in Code	Weak	Weak	C
Bielikova and Navrat (1998)	Weak	Schemata	Weak	No	Prolog
de Raadt (2008)	Weak	Plans in Code	Weak	No	C
Glaser et al. (2000)	No	No	Yes	Yes	ML
Felleisen et al. (2004)	No	No	Yes	Yes	Scheme
Caspersen and Kölling 2009	No	No	Yes	Yes	Java
This Research	Yes	Plans in VPE	Yes	Yes	Visual Programming Language

Table 1: Comparison of related approaches

1986), and a *plan* (Spohrer, Soloway, and Pope 1985) corresponds to a fragment of code that performs actions to achieve a goal. Goals and plans are key components in representing problems and solutions (Soloway 1986).

In the 1980s, Soloway and his colleagues (Letovsky and Soloway 1986, Soloway 1986, Spohrer *et al.* 1985) discovered that experts have strategies to solve problems using their library of plans. They advocated teaching these strategies and plans explicitly so that novices could have sufficient instructions on how to “put the pieces together”. Concurrently, they proposed (Soloway 1986) to use a goal and plan “language” for novices to explicitly construct their own plans. Moreover, a tool, GPCeditor (Guzdial *et al.* 1998) was created that supported novices to write a program based on the decomposition and composition of goals and plans. However, there was not a detailed process to support the composition of pieces of plan code, and, furthermore, the tool’s evaluation did not clearly demonstrate a significant advantage.

Similarly, pedagogical programming patterns were advocated by Porter and Calder (2003) by using small programming pieces in teaching novice programmers. Once again, a tool, ProPAT, was inspired by the idea of programming patterns (de Barros *et al.* 2005) allowing novices to insert code from the pedagogical patterns. However, there was weak support for how to apply these patterns in the goal analysis. Furthermore, there was not a detailed process of programming. Earlier, Bieliková and Návrat (1998) attempted to teach students a set of standard structures (or program schemata) as well as a method for how to apply them, but there was only a weak description of the goals achieved by the schema, and testing and debugging were not supported.

Recently, the goal and plan concepts have been taught as programming strategies in curricula by de Raadt (2008). Each strategy was also called a plan, which was basically pattern-like program code with examples. This approach attempted to integrate plans to build the program code after explicitly introducing goals and plans. However, it lacked a detailed process of programming development from goals to program via plans. Firstly, there were no clear guidelines for performing goal analysis. Secondly, there was no well defined process for merging plans. Thirdly, debugging and testing were excluded from the strategies in the program implementation.

As we have seen, a number of approaches that have used goals and plans have failed to provide detailed processes. On the other hand, there are approaches that have provided detailed processes, but they tended not to

use the concepts of goals and plans. For example, “Programming by Numbers” (Glaser, Hartel, and Garratt 2000) provides a clear process to create the smallest components of functions. It breaks the programming process into a series of well-defined steps and gives students a way of “programming in the small”. A similar, but more detailed, systematic design method was applied by Felleisen *et al.* (2004) to produce well-specified intermediate products in a stepwise fashion called “TeachScheme”. However, although both approaches emphasize the detailed process, the goal and plan concepts are not included. Additionally, both approaches are data-driven and more suited to functional programming languages than to mainstream procedural languages. More recently, a stepwise improvement process, STREAM (Caspersen and Kölling 2009), was developed as a conceptual framework with six major steps for teaching novice object-oriented (OO) programming together with five rules for implementing OO methods in order to break the task into smaller steps.

Table 1 summarizes the related work in terms of whether it uses goals and plans, whether a detailed process is provided, and whether testing and debugging are supported. As can be seen, existing work tends to either use goals and plans, but not provide a detailed process for guiding novices; or it provides a detailed process but does not use goals and plans.

3 Representing Goals and Plans

In order to be able to develop designs in terms of goals and plans, we need to have a way of representing them. Using a VPE such as Scratch, it is thus crucial to develop a visual notation for goals and plans.

3.1 The Visual Notation of Goals

Every program has a certain number of goals to be accomplished. Our visual notation for goals distinguishes between three basic types of goals: Input, Process, and Output (see Figure 1). A simple program might have one goal of each type, and achieve these goals in sequence. More generally, a program may have multiple goals of each type, and these goals can be combined using a mixture of sequential and parallel composition. This order is indicated graphically (see Figure 2), where the order of goal processing is left-to-right, and where goal decomposition is indicated by nesting. For example, Figure 2 shows that Goal 1 (an input goal) is followed by an unnamed goal which has been decomposed into three processing sub-goals: Goal 2 and 3 (which are achieved in

parallel), followed by Goal 4. These three goals are then followed by Goal 5 (an output goal).

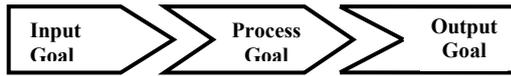


Figure 1. Notation for input, process and output goals

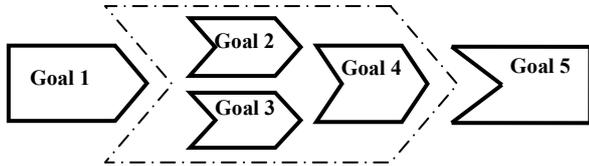


Figure 2. Notation for goal ordering

3.2 The Visual Notation of Plans

The visual plans are built up in order to implement the blueprint of the goals, following the metaphor of a network of plans that communicate using dataflow.

Each goal corresponds to a plan. Accordingly, there are three types of plans: input plans, process plans, and output plans. An input plan inputs data and then produces an output dataflow. Conversely, an output plan consumes the dataflow and displays it. A process plan consumes its input dataflow, and then processes it to produce a new dataflow in order to achieve the goal of the process.

Each plan is represented by a *plan block*. Plan blocks (see Figure 3) are constructed using BYOB², an extension to Scratch that allows you to Build Your Own Blocks. Each plan block has a unique name and parameter(s), including named “plan ports” (either “in” or “out”), which are used to connect the dataflow between the plans. A plan block also has an internal definition (not shown in Figure 4) which is just a BYOB process, constructed from standard BYOB constructs and the provided scaffolding blocks (discussed below). A group of existing plan blocks are developed and called a *visual plan library*. The visual plan library supports novices in designing and implementing their program (see Figure 4).

A set of visual plans can be linked by their dataflow ports to achieve a given set of goals (see step 2 in Figure 6). The set of plans is viewed as a network where items of data “flow” from one plan to another. The advantage of this “plan network” model is that because plans are conceptually concurrent, there is no need to worry about the correct sequencing of plans. This allows the plan network to be executable even before the plans have been merged (see Section 4.2).

In order to allow the plan network to be executable we have developed some special blocks which send and receive data to and from other plan blocks, and which indicate the linkage between the plan blocks. These blocks are referred to as “*scaffolding blocks*” and they are used to define the linkages between plan blocks (and also to send and receive data between plan blocks – see Section 5). For example, a “Begin Links” scaffolding block (See Figure 5 and step 2 in Figure 6) is followed by a number of “Link” scaffolding blocks, which capture the links between the plan blocks. Each “Link” block links two plans by its two parameters. The first parameter indicates the out-port of a plan and the second parameter links this to the in-port of

another plan. An “End Links” block indicates the end of the collection of “Link” blocks. A further example of using scaffolding blocks in the process of converting the plan-based solution into a single BYOB program will be illustrated in Section 4. The method of applying scaffolding blocks to construct new plan blocks will be described in Section 5.

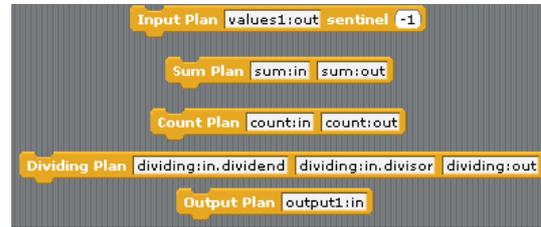


Figure 3. Example of plan blocks developed in BYOB



Figure 4. Example of plan library developed in BYOB



Figure 5. Scaffolding blocks for plan block linkage developed in BYOB

4 The Process of Programming

The process of programming from the visual notation of goals and plans to the final program consists of five steps: (1) analysing goals; (2) designing a network of plan blocks; (3) expanding the plan blocks; (4) merging the expanded plan details; and (5) simplifying the merged details (see Figure 6). The process is illustrated using the following example, which was also used by both Soloway (1986) and de Raadt (2008) to analyse goals and plans:

Write a program that will read in integers and output their average. Stop reading when the value -1 is input.

² <http://byob.berkeley.edu>

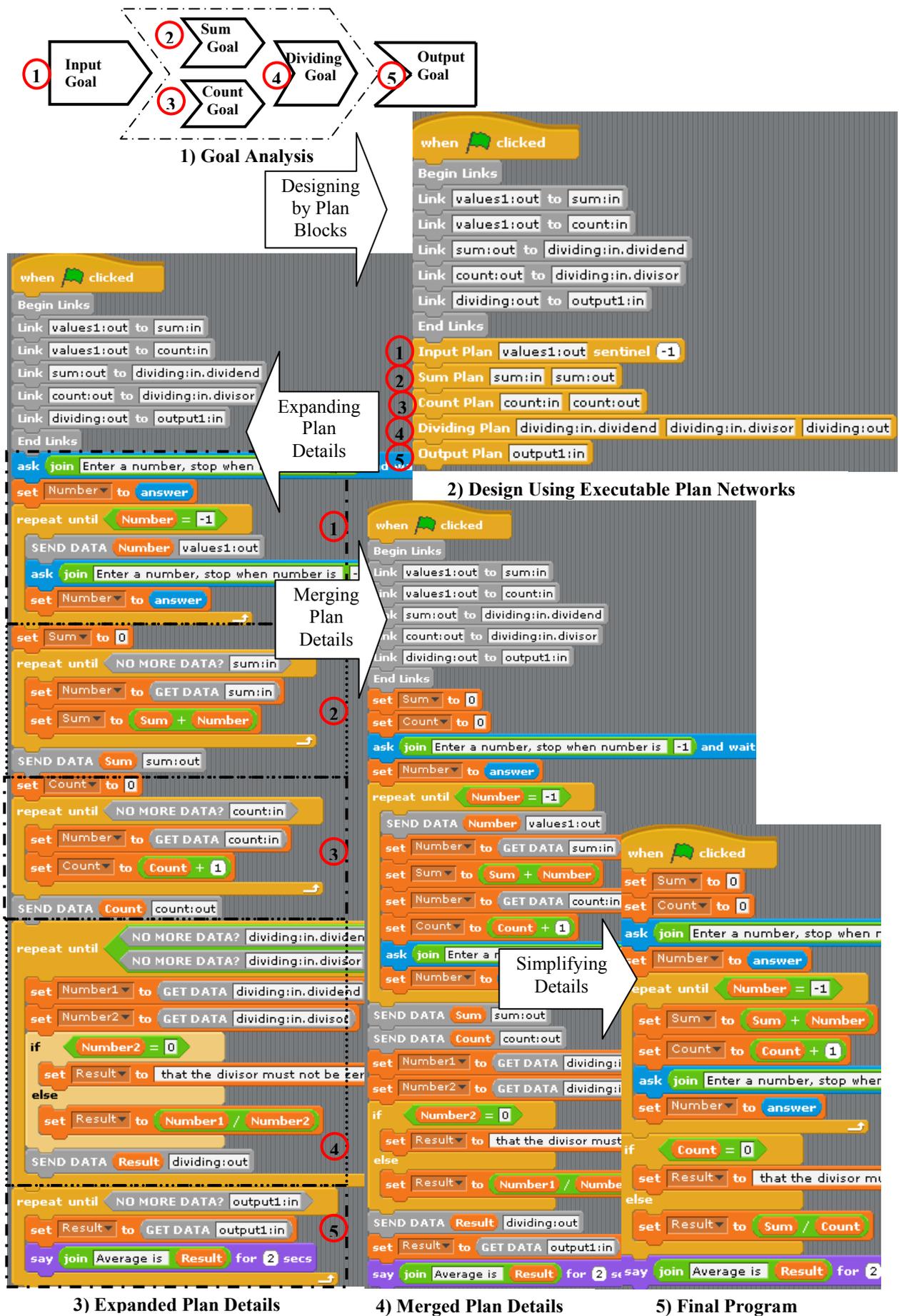


Figure 6. The process of programming from visual notation of goals and plans to the final program

4.1 Analysing Goals

Analysing goals involves identifying what goals the program needs to achieve. Typically these include at least one input, one output, and some number of processing goals, some of which may have sub-goals.

Identifying goals is done by a process of problem decomposition. For example, in order to achieve the goal of displaying the average of a sequence of values, one needs to read input, compute the average, and then display it. These correspond to goals. The second goal, computing the average, can be decomposed further resulting in five goals (see step 1 in Figure 6). The first goal is to input the values to be averaged. The program then needs to obtain both a “sum” and “count” from these values. It then accomplishes the goal of dividing the “sum” by the “count” giving the valid result “average”. Finally, the result will be displayed. Note that the goal notation is used purely as a design aid: we have not (yet) extended the BYOB tool to provide support for the goal analysis step.

4.2 Designing a Network of Plan Blocks

The network of plans is derived from the goal analysis by creating a plan for each goal, and deriving the dataflow based on the data requirements, in accordance with the goal ordering. For example, the dataflow starts from the “input” goal. The “sum” goal is after the “input” goal, and therefore data can flow from the “input” plan to the “sum” plan. Meanwhile data can also flow from the “input” plan to the “count” plan. After both goals of “sum” and “count” are achieved by the corresponding plans, the goal of “average” can be completed by a “dividing” plan, consuming dataflow from both the “sum” and “count” plans. Finally, the goal of displaying the “average” is reached by the “output” plan according to the dataflow from the “dividing” plan.

Based on the goal analysis, five plan blocks will be applied to build up a “plan network” in order to achieve these goals (see step 2 in Figure 6). The resulting design is captured using plan blocks and link blocks, with the plan blocks being listed in sequence, based on the ordering of the goals identified in the goal analysis (since the goals may not be in a strict sequence, the ordering is a partial one). A set of link blocks is then used to capture the links between plan blocks from out-port to in-port, based on the order of dataflow between them. For example, “Link values1:out to sum:in” links the out-port of the “input” plan to the in-port of the “sum” plan. Meanwhile, “Link values1:out to count:in” also connects the out-port of the “input” plan to the in-port of the “count” plan. Therefore, the dataflow from the “input” plan is copied and made available to both “sum” and “count”. Conversely, the dataflows from the out-ports of both “sum” and “count” plans are joined together into the “dividing” plan by the next two link blocks. Finally, the result is linked from the “dividing” plan to the “output” plan.

As noted earlier, a key feature of our approach is that this design can be tested by running the program containing the scaffolding blocks. Furthermore, the scaffolding blocks inside the plan block provide feedback if the parameters in the “Link” block do not match the names in the plan block.

4.3 Expanding the Plan Blocks

Expanding plan blocks means replacing each plan block with the defined block details within it (see step 3 in Figure 6). Whereas the previous two steps, goal analysis and designing a network of plan blocks, require human thought and creativity, this step is purely mechanical and could be automated (although adding this support to BYOB is future work).

The result of expanding the plan blocks is also executable and testable, which allows novices to obtain feedback rather than having to wait until the end of the process. Visualisation of dataflow through the plan-ports can be seen on the BYOB “Stage” by stepping the blocks to provide feedback.

4.4 Merging the Plan Details

Merging plan details aims to combine the plan details of the different plans into one program that does not make essential use of the scaffolding blocks (although they are still present; see step 4 in Figure 6). The merge principle is presented as a set of rules and also demonstrated to students using examples. Although this process is clearly defined, it is somewhat complex, and providing tool support for the merging process is a key direction for future work.

The basic steps of merging the plan details are:

1. collect all the blocks that initialise variables, and put them at the start of the program, i.e. immediately after the “End Links” block (e.g. the first two statements, “set Sum to 0” and “set Count to 0”, of the merged plan details in Figure 6, step 4);
2. next, put together the statements (including input statements) that initialise variables that are used in loop conditions (e.g. the 3rd and 4th statements, “ask” and “set”, in the merged plan details in Figure 6, step 4);
3. merge two loops when the second loop is “driven” by output from the first loop. Specifically, when the first loop outputs values via a port that is linked to an input port of a second loop, and the second loop has the structure “repeat until NO MORE DATA?(in-port)”. For example, the body of the “sum” Plan has a loop that gets data from an input port (“sum:in”) which is linked to the output port of the “input” plan (“values1:out”). In other words, after the first loop from the “input” plan sends data to the out-port “values1”, the first loop from “sum” plan gets data from the linked in-port “sum:in”. In other words the output from the first loop “drives” the second loop. Thus, the loop from the “input” plan is merged with the loop from the “sum” plan. When merging loops, the statements in each of the loops are kept in order, with the statements from the second loop being placed after those from the first loop. An exception is (input) statements that affect variables used in the loop condition: these are put at the end of the loop. For example, the two “set” statements from the second loop (“set Number to GET DATA sum:in” and “set Sum to SUM +

Number”) are put into the first loop after the original statement “SEND DATA Number values1:out”. However, the input statements “ask” and “set” affect the loop condition, and so are put at the end of the merged loop. Similarly, the loop of the “count” plan is also merged.

- any loop which is driven by reading a dataflow that will only have a single value can be simplified by replacing the loop with its body (e.g. the fourth and the fifth loops).

Once more, the result of merged plan details is executable and testable with the values of variables being visible in the BYOB stage.

4.5 Simplifying the Merged Details

The last step in the process is to simplify the merged details. This is done by combining variables that deal with the same data but have different variable names, and then removing all the scaffolding blocks to obtain the final program. The steps are:

- if a variable’s value is sent on an output port, and subsequently read from the linked input port into another variable, then the second variable should be consistently replaced with the first one. For example, consider the sequence “SEND DATA Sum sum:out”, and “set Number1 to GET DATA dividing.in.dividend”. Because the two ports are linked, the value of Number1 is taken from Sum, and so Number1 should be consistently replaced with Sum. Similarly, variable Number2 is replaced by variable Count;
- remove the use of ports and the associated scaffolding blocks (e.g. blocks from “Begin Links” to “End Links”, all the “SEND DATA” blocks, and all the blocks containing block “GET DATA”). This results in the final program shown in Figure 6.

5 Build Your Own Plan Blocks

The process presented in the previous section can be used by a novice programmer to develop a program using the plan blocks in the visual plan library. After the user has become familiar with the process of programming from goals and plans by using the existing plan blocks in BYOB, the next stage is to have them build their own plan blocks to add to the plan library. Since the input and output plan blocks can be used in most situations, the following only describes (very briefly) how to build process plan blocks. These are built using Scratch’s constructs (control flow, assignment, etc.) and three scaffolding blocks (see Figure 7): **NO MORE DATA? (in-port)** which returns a “true” or “false” value to indicate whether there is any more data from the “in-port”; **GET DATA(in-port)** which returns a data value retrieved from the given “in-port” of the current plan; and **SEND DATA (value, out-port)** which sends the data value to the linked plan(s) on the “out-port”.

Creating a process plan block which produces a single value result can be done by following a pattern (see Figure 8). Firstly, it must have at least two parameters, “in-port” and “out-port”, which have default values “[plan name]:in” and “[plan name]:out”. Secondly, the plan block normally starts with the initialization of variables. Thirdly, a “repeat until” loop retrieves a sequence of values from

the in-port. The loop condition is whether there is more data from the in-port using the scaffolding block “**NO MORE DATA?**”. Fourthly, inside the loop body, the first thing is to get a value from the in-port using the scaffolding block “**GET DATA**” and assign it to the variable initialized before the loop. Then it processes the value according to the algorithm of this plan, e.g. accumulating a running total for a Sum plan block. Finally, it sends the result to the out-port using the scaffolding block “**SEND DATA**”. The example of the Sum Plan block follows this pattern, and is built as in Figure 9.

Similarly, when a process plan block produces a sequence of values to its out-port and each of the output values is directly related to a value from the in-port, the process of building the plan is similar to that above, but the last step of sending a value to the out-port is inside the loop body. For example, the Even Number Plan block is created to produce even numbers from a sequence of values as in Figure 10.



Figure 7. Scaffolding blocks for constructing new plan block developed in BYOB

Plan Name (in-port = [name of plan]:in), (out-port = [name of plan]:out)	
Initialize variable	
repeat until < NO MORE DATA? (in-port) >	
	set (variable) to (GET DATA (in-port))
	Process variable according to the algorithm
SEND DATA (variable, out-port)	

Figure 8. A pattern for a process plan

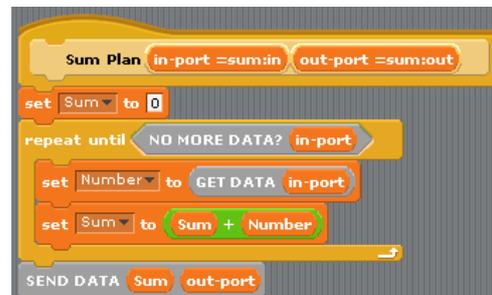


Figure 9. The sum plan details

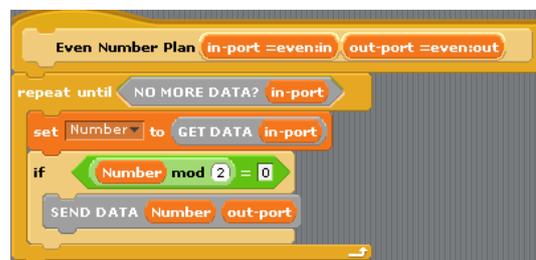


Figure 10. The even plan detail

6 Evaluation

The approach described so far has been evaluated by having students use it. This section describes the evaluation setting, data collected, analysis method, and results.

6.1 Experimental Setting

The first author taught novice programming at a polytechnic (equivalent to TAFE in Australia) from 1997 to 2009. This teaching used a traditional approach where the constructs of a programming language (C++ before 2000, and Visual Basic from 2000) were introduced and then flowchart and pseudocode were taught as techniques for program design. Desk-checking, testing and debugging were also taught. Although the author had introduced various teaching innovations over the years (Hu 2004), these pre-date 2006, and during the period which we consider (2006-2009), there were no significant changes to the teaching content or method.

In 2011 the same author taught this course again using the proposed approach described in this paper. The first part of the course was still traditionally taught in terms of syntax, pseudocode, flowchart, desk-check, testing and debugging, but using BYOB. However, the second part followed the ideas proposed in this paper: teaching the ideas of goals and plans, and the process presented in Sections 4 and 5. The two parts were separated by a mid-term examination. After the mid-term examination the experimental method was introduced into the curriculum for four weeks, three hours per week, using the framework of the plan library and scaffolding blocks in BYOB.

In both 2011 and in previous years students were heterogeneous with different age groups and academic background. They were taught interactively in a small class (held in a lab class) with a mixture of theory and practice. The assessments included converting from flowchart to pseudocode and also from pseudocode to flowchart as well as programming. Similar programming questions were used from year to year, such as calculating the sum and (positive or negative) count, or the average of a sequence numbers.

In order to assess programming ability we collected the answers to the programming question in the final examination (and, for 2011, the mid-term test as well). We only considered the programming question (the examination also had other questions that did not assess the students' ability to develop a complete program, the students could have done poorly on the programming question, but still may have done well on the examination overall). In all years (2006-2009, and 2011 in the mid-term and final examinations) the programming question was done on a computer, using a programming environment, rather than on paper. Note that a standard practice in polytechnics is that students who fail the examination are given a chance to re-sit the examination (some conditions apply). Where students took this opportunity, we only took results from their first attempt.

The students' answers on the programming question were re-marked using a common marking rubric. The criteria used were:

1. identifying all the variables correctly, for example when calculating an average, important variables included "sum" and "count";
2. correctly using fragments of key code, for example having code to count the number of values entered (but for this criteria we assessed the presence of essential code fragments, without

requiring them to be combined correctly);

3. combining code fragments correctly; and
4. the final program being tested and bug free.

Note that these criteria are cumulative in the sense that having a tested and bug free final program required the presence of correctly combined code fragments. A final score out of 100 was calculated by summing these four criteria. The weighting used was 10 for the first criteria, 40 for the second, 30 for the third, and 20 for the final criteria.

The distribution of students and their scores on the re-marked programming question in the examination is shown in both Table 2 and Figure 11. The wide range of performance, including both very low and very high scores, is typical of this sort of paper. The score results also support Caspersen and Kölling's argument (2009) of ending up with "two groups of students" with or without their own process. The median and average (Table 2) are therefore not particularly useful, and the actual scores for the re-marked programming question (Figure 11) give a better picture of the performance of students in each year.

Method	Year	Number of Students	Average Scores	Median value
Conventional Method	2006	13	33.3	18
	2007	16	53.8	82.5
	2008	13	36.8	0
	2009	8	39.4	22.5
	Mid-term 2011	7	52.4	40
Experiment Method	2011	8	84.8	100

Table 2: Summary of Results

We note that there appears to be a ceiling effect: for the 2011 experimental method a number of students scored 100%. This is not the case for the other cohorts, who were assessed using an equivalent instrument. The impact of this ceiling effect is therefore that it reduces the difference between the experimental group and the other groups. In other words, if we had used a measurement instrument that did not exhibit this ceiling effect, we would expect to see a *more* significant difference between the 2011 experimental cohort and the other cohorts.

6.2 Statistics Methodology

The Kruskal-Wallis one-way analysis of variance by ranks (Kruskal and Wallis 1952) is a statistical test for measuring the likelihood that a set of samples all come from populations with the same probability distribution. It is a non-parametric test, which means that it does not make any assumptions about the shape of the underlying probability distribution. This is important for our study because the performance of novice programmers is known to not follow a normal distribution, so parametric analysis of variance techniques are not valid (especially as our sample sizes are small). The Kruskal-Wallis test is an "omnibus" test, which means that a single test is used to compare a number of statistics (the medians of the samples in this case). It is common to follow a significant omnibus test (i.e. if the null hypothesis is rejected) by a family of pairwise tests to gain more precise information about the

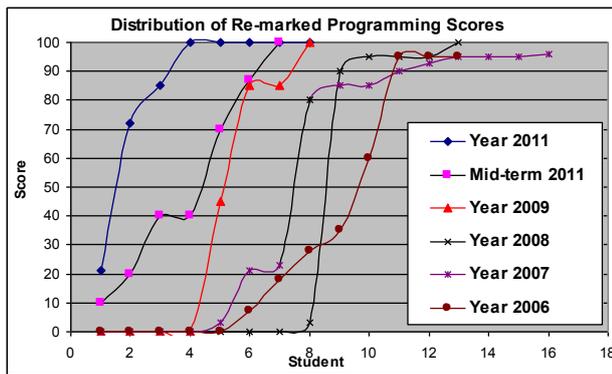


Figure 11. Distribution of results

causes of the significant difference. We do this using the Mann-Whitney U test, but we only consider the four comparisons between the samples from 2011 and each of the earlier years. This is because 2011 is the year in which the intervention we wish to measure was applied.

In many cases these pairwise tests are “post hoc” tests that were not pre-determined by the experimental design, and in this case it is necessary to adjust the individual test threshold values to reduce the overall chance of obtaining any false positive results (“type 1 errors”) across the family of tests (since performing more tests increases the chance of obtaining significant results by pure chance). The adjustment can be done using various forms of the “Bonferonni correction”. We use the Holm sequential Bonferonni method (Holm, 1979), which is less conservative than some other forms of adjustment. While we have used this technique (and still obtained a significant result), we note that our pairwise tests are not post hoc tests - they follow naturally from the nature of our experiment, and could arguably have been done without this adjustment. That would have resulted in a higher level of significance. Both Kruskal-Wallis test and Mann-Whitney U test tests have the same assumptions:

1. The variables of student examination scores have a continuous distribution. Although our scores are not continuous, they are fine grained (out of 100);
2. The measurement scale of assessment score is at least ordinal, which clearly is the case here;
3. Samples of student scores are independent from each year. We ensured this by checking for students who enrolled in the same course more than once, and only considered them the first time they participated in the course.
4. Samples of each year’s scores (i.e. the different populations) are “of approximately the same form” (Kruskal and Wallis 1952, page 585). In other words, these samples come from populations with the same shapes and spreads of distribution, or from populations with identical medians. Since there were no significant changes to the course content in 2006-2009, we have no reason to expect the distributions to be different, and the distribution of scores (Figure 11) do not appear to be significantly different.

6.3 Hypotheses and Results

Our analysis aimed to determine to what extent the new

approach for teaching programming made a difference. However, in order to draw conclusions about any difference between the 2011 results and results from earlier years being due to the new teaching approach we need to rule out alternative explanations.

In order to do this we developed two additional hypotheses. Firstly, we test the hypothesis that there is no significant variance within the years 2006-2009. If this hypothesis holds, then it suggests that differences due to variances in the cohort across years, or (slightly) different assessment questions are not significant. Secondly, we hypothesise that there is no significant difference between the earlier years and the 2011 mid-term test. If true, this suggests that any difference between the 2011 final results and earlier years are not due to differences between the 2011 cohort and the cohorts in earlier years. It would also suggest that the use of BYOB was not, of itself, sufficient to explain any difference. This finding would support Lister (2011), who argued that using Scratch and Alice will allow novice programmers to make better initial progress compared to using other languages, but that without a “pedagogical rethink of what should happen after these tools”, there would still be issues when students are required to perform tasks that require transitive inference, such as realising that checking whether an array is sorted is equivalent to checking whether each pair of consecutive items are in order. In other words, visual programming languages may support novices to start programming. However, there must be a well defined pedagogical method to help novices bridge the gap to becoming experts. Finally, we hypothesise that there is a significant difference between the final results in 2011 and the results in previous years.

Hypothesis 1: The null hypothesis is that the examination scores from 2006 to 2009 come from populations with identical “locations”. In other words, the median scores from each year are expected to not be significantly different.

The p-value of the examination scores from 2006 to 2009 by Kruskal-Wallis Test is 0.689 (> 0.05). It means there is not enough evidence to reject the null hypothesis. In other words, there are no significant differences in median scores in the past (2006-2009). This suggests that the examination results are similar in past years when teaching by the conventional method and that any differences of student cohort and variations of examination questions are not significant.

Hypothesis 2: The null hypothesis is that the mid-term examination scores of year 2011 and the examination scores from 2006 to 2009 also come from populations with identical “locations”. In other words, the median scores in the past years and year 2011 mid-term are expected to not be significantly different.

The p-value from the student scores in the mid-term examination of year 2011 and those from 2006 to 2009 is 0.603 (> 0.05). This suggests that the examination results are similar in each year when still teaching by the conventional method despite differences of student cohort, variances of examination questions, and inconsistencies of computer languages (VB vs. BYOB).

Hypothesis 3: The null hypothesis is that the final examination scores from year 2006 to 2009 and year 2011

come from populations with identical “locations”. In other words, the median scores of year 2011 based on the experimental method and years in the past based on the conventional method are hypothesised to not be significantly different.

The p-value from the student scores in the past examinations (2006 - 2009) and those from the final examination in 2011 is 0.031 (< 0.05). This shows a significant difference between the median examination score in 2011 and the examination scores in previous years, and provides evidence for rejecting Hypothesis 3. Since there is no significant difference in the past (Hypothesis 1), it implies that the difference comes from year 2011. Furthermore, since there is no significant difference between the past and the 2011 mid-term examinations (Hypothesis 2), it suggests that the difference comes from changing to different teaching methods after the 2011 mid-term examination.

Since the above tests did not clearly indicate *which* year(s) causes the difference, we conducted paired comparisons. Because the Kruskal-Wallis Test on the data from all years except 2011 showed no significant difference (Hypothesis 1) we only considered paired comparisons between 2011 and other years (2006 -2009). We used Holm’s sequential Bonferroni method (Holm, 1979) in order to reduce the chance of any type 1 errors. With this method the p-value (from smallest to largest) by Mann-Whitney U Test for each paired comparison needs to be smaller than its threshold p-value to be significant, where the threshold p-value is divided by (C-i+1), making the test more conservative. For example, the first test, which is the one with the lowest U-test p-value (comparing 2011 and 2006) has a threshold p-value of 0.05/4; the second comparison has a threshold of 0.05/3, etc. The results of these Mann-Whitney tests (see Table 3) indicated significant differences of examination scores between the year 2011 and each individual year from 2006 to 2009.

Paired Comparisons (C= 4)	Threshold p- value $0.05/(C - i + 1)$	p-value by U test
between 2011 to 2006	0.013	0.003 (<0.013)
between 2011 to 2008	0.017	0.01 (<0.017)
between 2011 to 2007	0.025	0.021 (<0.025)
between 2011 to 2009	0.05	0.025 (<0.05)

Table 3: Comparing Paired Examination Scores

To summarise, through the analysis of past examination results we have seen a statistically significant improvement in student performance using our new approach. We have provided evidence that the difference is not due to variation in the cohort, in the examination questions, or due to the use of BYOB.

7 Conclusion

Our research suggests that the experimental teaching method which combines using goals and plans, a well-defined process, and a visual notation, has the potential to significantly improve learning of programming skills. A key feature of our approach is that we provide a detailed process that guides novices through the process of developing a program, using goals and plans. Another key feature that distinguishes our approach from other work is that our representation for goals and

plans is integrated into a (visual) programming language, and that this integration is done in such a way as to allow an intermediate program to be executable. Once plans are defined and linked, the resulting program can be executed, even though the plan bodies have not yet been merged to produce a final program. We see this as a significant advantage for a number of reasons. Firstly, it provides earlier feedback, and supports testing and debugging. Secondly, plan merging is known to be difficult (Soloway 1986), and by allowing novices to obtain feedback on their unmerged designs, they can improve their design without having to perform plan merging. Finally, it allows novices to distinguish between errors in their design and errors introduced by faulty plan merging.

A limitation of the evaluation is that we considered the new approach as a package. While this makes sense in that it is the *combination* of factors that makes the approach effective, it is possible that some factors are less important than others. For example, students in 2011 had a plan library provided. This clearly assists with completing a programming task, and we cannot say to what extent the improvements in student performance were due to this factor. Another limitation of our evaluation is that we only used a programming question in an examination to measure performance. We argue that an examination is an appropriate choice because it is conservative: it tends to underestimate ability (due to time constraints), and it eliminates the possibility that exists in assignments that students obtained significant assistance from peers, family, friends, or tutors. We also did consider assignment results, and found that there were no significant differences between 2011 and earlier years. Another limitation of the evaluation is that we had only a limited number of students in each year. Future work could include evaluating this framework with more students. Other potential issues with the evaluation are that the course was taught by an author of this paper, who might be expected to be enthusiastic about the new approach. We argue that while this is certainly true, the author was equally enthusiastic about their past teaching and that when conducting teaching in 2006-2009, the approach described in this paper had not yet been developed, or even conceived. Finally, the students in 2011 were aware that they were being taught using a modified experimental method (since they had to sign ethics approval forms), but any form of Hawthorne effect would be expected to apply to the whole course, including performance in the mid-term test, since students did not know which part of the course was traditional and which was novel.

There are a number of directions for future work. At the moment we have defined visual notation for both goals and plans, but only the visual notation for plans has been integrated into Scratch. One area for future work is therefore completing the integration of the representation of goals into Scratch. Additionally, although the plan merging process is well-defined, it is somewhat complex, and one key area for future work is therefore how to support novices in performing plan merging. Because we want novices to eventually move away from using our framework, it is important that this support *not* be in the form of a “wizard” that does the merging of plans without the student gaining any insight into the merging process.

8 References

- Bieliková, M. and Návrát, P. (1998): Learning programming in Prolog using schemata. *ACM SIGPLAN Notices*, **33**(2): 41-47.
- Caspersen, M. and Kölling, M. (2009): STREAM: A First Programming Process, *Transactions on Computing Education (TOCE)* **9**(1), 4:1-29, ACM
- de Barros, L. N., dos Santos Mota, A. P., Delgado, K. V. and Matsumoto, P. M. (2005): A tool for programming learning with pedagogical patterns. *Proc. 2005 OOPSLA workshop on Eclipse technology eXchange*, ACM, 125-129.
- de Raadt, M. (2008): Teaching programming strategies explicitly to novice programmers. *Doctoral Thesis*, School of Information Systems, University of Southern Queensland.
- du Boulay, B. (1989): Some difficulties of learning to program. Chapter In: *Studying the Novice Programmer*, E. Soloway & J. C. Spohrer, Eds. Lawrence Erlbaum Associates, Hillsdale, NJ, ISBN-0805800034, 283-299.
- Felleisen, M., Findler, R. B., Flatt, M. and Krishnamurthi, S. (2004): The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, **14**(1):55-77.
- Glaser, H., Hartel, P. H. and Garratt, P. W. (2000): Programming by numbers: a programming method for novices. *The Computer Journal*, **43**(4):252-265.
- Guzdial, M., Konneman, M., Walton, C., Hohmann, L. and Soloway, E. (1998): Supporting programming and learning-to-program with an integrated CAD and scaffolding workbench. *Interactive Learning Environments*, **6**(1/2):143-179.
- Hu, M. (2004): Teaching novices programming with core language and dynamic visualisation. *Proc. the 17th Conference of the National Advisory Committee on Computing Qualifications* (Christchurch, 6 - 9 July), 95-104, New Zealand: NACCQ
- Kay, J., Barg, M., Fekete, A., Greening, T., Hollands, O., Kingston, J. H. and Crawford, K. (2000): Problem-based learning for foundation computer science courses. *Computer Science Education*, **10**(2):109-128.
- Kelleher, C. and Pausch, R. (2005): Lowering the barriers to programming. *ACM Computing Surveys*, **37**(2):83-137.
- Kruskal, W. and Wallis, A. (1952): Use of ranks in one-criterion variance analysis, *Journal of the American Statistical Association*, **47**(260):583-621.
- Lahtinen, E., Ala-Mutka, K. and Järvinen, H. M. (2005): A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin*, **37**(3):14-18.
- Letovsky, S. and Soloway, E. (1986): Delocalized plans and program comprehension. *IEEE Software*, **3**(3):41-49.
- Lister, R. (2011): Programming, Syntax and Cognitive Load, *ACM Inroads*, 2011 June, **2**(2):21-22
- Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K. and Seppälä, O. (2004): A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, **36**(4):119-150.
- McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B. D., Laxer, C., Thomas, L., Utting, I. and Wilusz, T. (2001): A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, **33**(4):125-180.
- Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M. and Paterson, J. (2007): A survey of literature on the teaching of introductory programming. *Proc. Working group reports on ITiCSE on Innovation and technology in computer science education*, ACM.
- Porter, R. and Calder, P. (2003): A pattern-based problem-solving process for novice programmers. *Proc. Fifth Australasian Computing Education Conference (ACE2003)*, Adelaide, Australia. Greening, T. and Lister, R., Eds., ACS. CRPIT, **20**:231-238.
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J. and Silverman, B. (2009): Scratch: programming for all. *Communications of the ACM*, **52**(11):60-67.
- Robins, A., Rountree, J. and Rountree, N. (2003): Learning and teaching programming: A review and discussion. *Computer Science Education*, **13**(2):137-172.
- Rößling, G., Joy, M., Moreno, A., Radenski, A., Malmi, L., Kerren, A., Naps, T., Ross, R. J., Clancy, M., Korhonen, A., Oechsle, R. and Iturbide, J. Á. (2008): Enhancing learning management systems to better support computer science education. *SIGCSE Bulletin*, **40**(4):142-166.
- Soloway, E. (1986): Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, **29**(9):850-858.
- Spohrer, J. C., Soloway, E. and Pope, E. (1985): A goal/plan analysis of buggy Pascal programs. *Human-Computer Interaction*, **1**(2): 63-207.
- Sykes, E. R. (2007): Determining the Effectiveness of the 3D Alice Programming Environment at the Computer Science I Level. *Journal of Educational Computing Research*, **36**(2):223-244.
- Winslow, L. E. (1996): Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin*, **28**(3): 17-22.

Toward a Shared Understanding of Competency in Programming: An Invitation to the BABELnot Project

Raymond Lister

Faculty of Engineering and
Information Technology,
University of Technology, Sydney,
Sydney, NSW, Australia
raymond.lister@uts.edu.au

Malcolm Corney

Faculty of Science and Technology,
Queensland University of
Technology,
Brisbane, Qld, Australia
m.corney@qut.edu.au

James Curran

School of Information Technologies,
University of Sydney,
Sydney, NSW, Australia
james.r.curran@gmail.com

Daryl D'Souza

School of Computer Science and
Information Technology,
RMIT University,
Melbourne, Vic, Australia
daryl.dsouza@rmit.edu.au

Colin Fidge

Faculty of Science and Technology,
Queensland University of
Technology,
Brisbane, Qld, Australia
c.fidge@qut.edu.au

Richard Gluga

School of Information Technologies,
University of Sydney,
Sydney, NSW, Australia
richard@gluga.com

Margaret Hamilton

School of Computer Science and
Information Technology,
RMIT University,
Melbourne, Vic, Australia
margaret.hamilton@rmit.edu.au

James Harland

School of Computer Science and
Information Technology,
RMIT University,
Melbourne, Vic, Australia
james.harland@rmit.edu.au

James Hogan

Faculty of Science and Technology,
Queensland University of
Technology,
Brisbane, Qld, Australia
j.hogan@qut.edu.au

Judy Kay

School of Information Technologies,
University of Sydney,
Sydney, NSW, Australia
judy.kay@sydney.edu.au

Tara Murphy

School of Information Technologies,
University of Sydney,
Sydney, NSW, Australia
tm@it.usyd.edu.au

Mike Roggenkamp

Faculty of Science and Technology,
Queensland University of
Technology,
Brisbane, Qld, Australia
m.roggenkamp@qut.edu.au

Judy Sheard

Faculty of Information Technology,
Monash University, Caulfield East,
Victoria, Australia
judy.sheard@monash.edu

Simon

School of Design, Communication,
& IT, University of Newcastle,
Ourimbah, NSW, Australia
simon@newcastle.edu.au

Donna Teague

Faculty of Science and Technology,
Queensland University of
Technology,
Brisbane, Qld, Australia
d.teague@qut.edu.au

Abstract

The ICT degrees in most Australian universities have a sequence of up to three programming subjects, or units. BABELnot is an ALTC-funded project that will document the academic standards associated with those three subjects in the six participating universities and, if possible, at other universities. This will necessitate the development of a rich framework for describing the

learning goals associated with programming. It will also be necessary to benchmark exam questions that are mapped onto this framework. As part of the project, workshops are planned for ACE 2012, ICER 2012 and ACE 2013, to elicit feedback from the broader Australasian computing education community, and to disseminate the project's findings. The purpose of this paper is to introduce the project to that broader Australasian computing education community and to invite their active participation.

Keywords: programming, objectives, assessment.

1 Introduction

It is very common for ICT degrees to incorporate a sequence of up to three programming subjects (also known as courses, papers, or units of study). Traditionally, these three subjects have formed part of the

Copyright © 2012, Australian Computer Society, Inc. This paper appeared at the 14th Australasian Computing Education Conference (ACE2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology, Vol. 123. M. de Raadt and A. Carbone, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

compulsory ‘core’ of ICT degrees, particularly software engineering degrees. There is certainly a great deal of variety between institutions as to what is covered in the subjects. While the first is typically thought of as an introduction to programming, the second might be a direct continuation of programming concepts, a data structures subject, a subject addressing program access to databases, and so on; and even more variation can be expected in the third subject. Nevertheless, it appears to be the case that many ICT degrees identify three specific subjects as an effective programming stream, and it is these three subjects with which this project is concerned. Despite the centrality of these three programming subjects, computing academics remain dissatisfied with the effectiveness of these subjects. Many students are also dissatisfied: in a widely discussed paper describing the educational ‘Grand Challenges’ in computing, McGettrick et al (2004) note that:

“educators cite failure in introductory programming courses and/or [student] disenchantment with programming as major factors underlying poor student retention”.

In programming subjects, as with most Australian university subjects, the semester begins in each classroom with the ritual distribution of the subject outline, which provides a brief description of the subject, the topics to be covered, and the assessment scheme. Although such outlines often run to many pages, the document can be ambiguous. For example, consider the following objective, taken from the outline of an introductory programming subject at one of the universities participating in this project:

On successful completion of this subject, the student will be able to ... Demonstrate a working knowledge of the basic constructs in the object-oriented language Java.

Which constructs are the “basic” constructs? What does it mean to have a “working knowledge”, and how does a student “demonstrate” it? Figure 1 shows an extract from Computer Science Curriculum 2008 (ACM/IEEE, 2008), which manifests numerous similar ambiguities.

Furthermore, while students may think of outlines as the contract between them and their teacher, outlines are conscripted into many roles. For example, outlines are presented to professional accreditation committees as evidence that the required subject matter is being taught. Mappings are sometimes made from subject outlines to a university’s graduate attributes. When a student moves to a new university and seeks credit for prior study, outlines are used to establish subject equivalence between the two universities. Very importantly, outlines are also a contract between teachers. In a three-semester sequence of programming subjects, for example, the second and third semester teachers rely upon the outline of the previous subject to define what students should know at the start of semester – and sometimes those teachers feel justified in complaining that the students cannot actually do what the previous subject’s outline says they can do.

If we ignore the relationship of a given subject to other subjects, be they different subjects at the same university or equivalent subjects at other universities, even a given

PF/Fundamental Constructs [core]

Minimum core coverage time: 9 hours

Topics:

- Basic syntax and semantics of a higher-level language
- Variables, types, expressions, and assignment
- Simple I/O
- Conditional and iterative control structures
- Functions and parameter passing
- Structured decomposition

Learning Objectives:

1. Analyze and explain the behavior of simple programs involving the fundamental programming constructs covered by this unit.
2. Modify and expand short programs that use standard conditional and iterative control structures and functions.
3. Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, and the definition of functions.
4. Choose appropriate conditional and iteration constructs for a given programming task.
5. Apply the techniques of structured (functional) decomposition to break a program into smaller pieces.
6. Describe the mechanics of parameter passing.

Figure 1: An extract from Computer Science Curriculum 2008 (ACM/IEEE, 2008)

subject varies over time. A change to the final exam is one of the most important yet subtle ways that a subject can change. If one loiters long enough in a departmental tea room around the time of semester when exams are being written, one will hear quite passionate complaints that Professor Bloggs has ‘watered down’ the final exam in a particular subject (e.g. by changing from free response to multiple choice). Changes to an exam often do not require changes to the subject outline or any other documentation, and can thus be made with little management oversight. Academics who teach downstream of that subject may not even be aware of the change until well after it has taken place.

1.1 The Relationship with Software Engineering

Software engineering as a discipline has wrestled with problems that are analogous to the pedagogical problems described above. A software engineering project usually begins with a long negotiation between the software developers and the various stakeholders. The negotiation culminates in a design document, often called a specification, which forms a contract between the software developers and the various stakeholders. Even the most comprehensive specification documents leave implicit some aspects of the proposed system, which are

remembered as shared understandings and oral agreements arising from certain meetings. After the implementation of the software begins, changes are inevitably made to the specification. Some changes are documented, while others remain implicit. Almost inevitably, not everyone is aware of, or happy with, some of the changes. While software engineering is by no means a solved problem, some aspects of the culture of software engineering can usefully be adopted to attack the problems in ‘pedagogical engineering’ described above. Thus academics with an ICT background bring a special perspective to specifying academic standards.

1.2 The BABELnot Project: Desired Outcomes

The above considerations led the authors to propose the BABELnot project. (See section 8 for an explanation of the name.) We successfully applied to the Australian Learning and Teaching Council (ALTC) for funds to support the project across the six participating institutions. The funded work of the project began in October 2011 and will continue until August 2013.

Our aim is to achieve consensus on a framework for describing learning outcomes in computer programming, specifically the teaching of programming in the first three semesters, and also on how to map between learning outcomes and exam questions. We understand that assessment in programming subjects is not restricted to written exams, and that some learning outcomes are often assessed by way of other forms of assessment such as assignments and practical tests; but these other forms of assessment are beyond the current scope of the project.

Our desired outcomes are:

- The creation of a bottom-up, action research approach to articulating learning outcomes, in the context of the first three programming subjects
- A culture of scholarly teaching in ICT, spanning institutional boundaries, with a discourse based in evidence rather than anecdote
- Exams that are a more valid and reliable indicator of student programming ability
- Better learning of programming by students
- Attraction and retention of more students to programming and software engineering

More specifically, our desired, measurable project deliverables are:

- A system for describing learning outcomes and assessment by written exam. A method for mapping between learning outcomes and exam assessment, applicable to the first three programming subjects
- The learning outcomes of the first three programming subjects, from at least the six participating universities, re-expressed within the system
- A document summarising an archive of exam questions, with meta-tags mapping the questions to the system, serving as examples for use by other academics
- Performance data from real students for a subset of the archived exam questions

In this project, learning outcomes will tend to be articulated in terms of a characterisation of suitable assessment tasks, for example,

“On successful completion of this subject, a passing student will be able to implement iterative algorithms on arrays, such as linear search, binary search and quadratic sorting algorithms, in approximately half an hour, without reference to external notes”.

Note that this is merely an illustrative example, not a recommendation of a standard to be adopted. Like Wright, Hadgraft, and Cameron (2010), it is not our intention to be prescriptive about what students at a particular institution should know, but rather to provide the framework within which academics at that institution might be prescriptive.

2 Background

This section reviews relevant prior work that motivated the development of this project and influenced the project’s design.

2.1 The BRACElet Project

A number of papers about the BRACElet project have been presented at past ACE conferences (e.g. Whalley et al, 2006). Work on BRACElet started in New Zealand in 2004. In 2007, the ALTC funded a fellowship project by Lister and Edwards to explicitly extend BRACElet into Australia (Lister & Edwards, 2010). The final BRACElet workshop was held in 2010 (Clear et al, 2011).

BRACElet recruited academics from multiple universities into an action research approach that involved the systematic collection of evidence from end-of-semester programming exams. As part of this process the project participants formulated ideas on where the problems lay for novice programmers, devised exam questions to test these ideas, and collected and analysed the data from the end-of-semester exams. This process was repeated several times. Contrary to the intuitions of many computing academics, the project participants found that students tend not to have problems with the low level ‘nuts and bolts’ of programming. Instead they have difficulties fitting the pieces together to see the larger picture – they ‘cannot see the forest for the trees’. Many traditional exam questions, however, largely test the novice programmer on the lower level nuts and bolts, and learning outcomes are often expressed in terms of these nuts and bolts.

Three workshops were held within Australia during the funding period of the ALTC fellowship. A total of 21 Australian academics, from 14 different Australian universities, either attended these workshops or actively participated in the project electronically. Academics from at least seven Australian universities have used end-of-semester exam questions that were designed as part of this project. The project has also attracted international attention, with academics from 14 universities in seven countries actively participating in data collection and analysis. During the ALTC Fellowship funding period, 26 project participants (co-)authored 16 published papers, further disseminating the outcomes of the project.

2.2 Course and Unit of Study Portal (CUSP)

The Course and Unit of Study Portal (CUSP) is a software product that was developed jointly by three faculties of the University of Sydney as a university-funded project to provide a common curriculum mapping framework for a diverse range of professional degrees across Engineering, IT, Architecture, Design, Urban Planning, and Health Sciences. CUSP is currently used at the University of Sydney for over 240 degrees and over 2,500 units of study across four faculties. (Note: the University of Sydney uses the term ‘unit of study’ for what some other universities call a ‘subject’ or a ‘course’, and the term ‘course’ for what some other universities call a ‘degree’ or a ‘program’.)

CUSP captures the representation of multiple sets of graduate attributes and accreditation competencies (named curriculum goals or curriculum goal frameworks) and maps these to the relevant degrees. Each degree structure is modelled into the system as a collection of core subjects plus the rules governing the selection of elective subjects. Each graduate attribute or accreditation competency is in turn mapped to each assessment and learning outcome within each subject of a degree. This design enables the CUSP system to generate reports that visualize the curriculum coverage for entire degrees against any of the curriculum goal frameworks attached. These reports in turn enable quick identification of any gaps in goal coverage or any sequencing problems in the degree structure and facilitate accreditation or other quality control review processes. This is described in greater detail by Gluga et al (2010).

Richard Gluga, a PhD candidate at the University of Sydney, is creating an enhancement of CUSP, known as ProGoSs – Program Goal Progression (Gluga et al, 2012), which can be used to map the detailed objectives for the programming fundamentals curriculum designed by the ACM/IEEE (2008). The extension is intended to support systematic design, modelling and monitoring of student progression as part of curriculum design using Bloom's Taxonomy (Bloom, 1956) and neo-Piagetian cognitive development theory (Lister, 2011). It also supports curriculum design by allowing for the specification of the level of achievement of both higher- and lower-achieving students, so that institutions can design a curriculum, and assess how well it is achieving its learning outcomes, with full regard to the range of achievement of the students who complete degrees.

2.3 Exam Question Classification

The aim of the Exam Question Classification project is to investigate the nature and composition of formal examination instruments used in summative assessment of introductory programming students, and the pedagogical intentions of the educators who construct these instruments. The project leaders presented their first draft of a classification scheme in a half-day workshop at the 2011 ACE Conference in Perth. On the basis of the feedback received from the 20 or so workshop participants, the project leaders revised their initial scheme. Subsequently, project members formed pairs and applied the revised scheme to analysing a total of twelve exams, from nine different universities in Australia, the UK, New Zealand, Finland and the USA. A paper on this

work was recently presented at the Seventh International Computing Education Research Workshop (Sheard et al 2011) and another is being presented at ACE 2012 (Simon et al 2012).

Properties encoded about an exam question in the current draft of the classification include type of question (e.g. short answer, multiple choice), topics examined (e.g. data types, loops, OO concepts, program design), type of skill required (e.g. knowledge recall, hand executing code, writing code, explaining code), and difficulty (high, medium, low).

Getting academics to agree on classifications of specific questions has not proved to be straightforward. For example, two of the project participants recently classified an introductory programming exam consisting entirely of multiple-choice questions. While computing academics are divided on the value and validity of multiple-choice questions (Shuhidan et al, 2010), they are nevertheless widely used (Simon et al, 2012). On the issue of degree of difficulty (high, medium, low) the two participants agreed independently on only one third of the multiple-choice questions. On skill required (e.g. knowledge recall, hand executing code, explaining code) they agreed independently on one quarter of the multiple-choice questions. It is hardly surprising that subject outlines and other documents are ambiguous, when two experienced teachers of introductory programming exhibit such a low level of agreement on a set of multiple-choice questions. Before there can be a substantive debate on the content and assessment of early programming courses, there needs to be greater consensus on a framework for the debate – a framework that this project aims to provide.

2.4 Neo-Piagetian Theory

Wright, Hadgraft and Cameron (2010) describe a dialectic in learning outcomes, with one part of the dialectic being a “*list of discrete outcomes or aspirational statements*” as opposed to the other part of the dialectic, “*threshold learning outcomes [that] reflect the way engineers and ICT professional approach, think and do their work*”. In this project we adopt a cognitive development perspective to transcend that dialectic.

Piaget developed a very well known constructivist theory about the different levels of abstract reasoning exhibited by people as they mature from child to adult. While classical Piagetian theory has been largely abandoned, neo-Piagetian theory has overcome many of the problems that led to that abandonment. The types of abstract reasoning are broadly the same in both theories; but in neo-Piagetian theory, people, regardless of their age, are thought to progress through increasingly abstract forms of reasoning as they gain expertise in a specific problem domain. Neo-Piagetians attribute the increasing abstraction in reasoning not to biological maturity but to an increase in the effective capacity of working memory, as the learner ‘chunks’ knowledge. Neo-Piagetian theory is not esoteric – the popular SOLO taxonomy (Biggs and Collis, 1982) is based upon neo-Piagetian theory.

In a paper presented at the 2011 Australasian Computing Education Conference, Lister (2011) proposed a way of applying neo-Piagetian theory to the learning of programming. He defined the development of

the novice programmer in terms of three neo-Piagetian stages. At a pre-operational stage, students can trace the changing values in a piece of code, but do not reason in terms of abstraction of that code. At a concrete operational stage, students can reason in terms of abstractions, but only in the context of specific code. At a formal operational stage, students can reason in terms of programming abstractions without recourse to explicit code examples. Lister's stage theory has already been adopted by CUSP participants at the University of Sydney, and empirical results from Queensland University of Technology (Corney et al, 2012; Teague et al, 2012) add support to the proposal.

3 Dissemination Strategy

There is very little point to this project, or to any other innovative, education-related project, if the outcomes of the project remain private to the direct project participants. In many respects, the success of any innovative, education-related project should be assessed by the degree of dissemination of the outcomes.

By 'dissemination', we do not simply mean the distribution of information via publications and seminars (although distribution of information is an essential component of a successful dissemination). For the authors of this paper, 'dissemination' is to be measured by the extent of adoption by others of the materials and techniques developed by the authors of this paper.

It is well documented that dissemination, as the term is used in this project, is difficult. Few innovative, education-related projects have succeeded at dissemination (Gannaway et al, 2011; McKenzie et al, 2005; Southwell et al, 2005 & 2010). To improve dissemination, the ALTC explicitly adopted a Dissemination Framework (ALTC, 2006), which has also guided the authors of this paper. Even with the untimely demise of the ALTC, this dissemination framework is likely to influence the design of Australasian education projects well into the future.

As advocated within the ALTC Dissemination Framework, this project has adopted an 'engaged' model for dissemination:

"involving consultation, collaboration and support for ongoing dissemination both during the project and after the project is completed"

Consequently, the dissemination of this project begins early in the project (indeed, it begins with the publication of this paper) and will continue throughout the project, based on proposed full-day workshops held at roughly six-monthly intervals in conjunction with major computing education conferences:

- ACE 2012 (January, Melbourne)
- ICER 2012 (August, Auckland)
- ACE 2013 (January, Adelaide)
- ITiCSE 2013 (June/July, Canterbury, UK).

While the first of these workshops is now confirmed, the other three will be subject to proposal and acceptance at the respective conferences. As the workshops also serve to define project stages and milestones, if any of the

proposals is not accepted, alternative dissemination mechanisms will be formulated.

The workshops will be open to all interested academics, and will probably not require a registration fee.

The budget allocation for dissemination and evaluation workshops includes a limited number of 'scholarships' that will pay the registration fee for ACE 2013, to be held in Adelaide. These scholarships will be awarded to people outside the project who contribute documents, data or other material that manifestly advances the project.

The ITiCSE working group reports are among the most influential and highly cited papers in computing education. Thus an ITiCSE working group in 2013 will maximise the potential for international dissemination.

3.1 Monthly Meetings

As part of the project, full-day meetings will be held each month in at least two of Melbourne, Sydney and Brisbane.

The project values collaboration, so these meetings are not necessarily closed, and researchers not currently involved in the project may be invited to attend them. However, while the six-monthly workshops are open to anyone even if they merely wish to observe, an invitation to a monthly meeting will be made on the assumption that the invitee will play an active and continuing role. For the types of active roles that are suitable, see Section 5.1, 'Rules of Engagement'. A person seeking to join the project under this arrangement may need to make an explicit time commitment. A 10% time commitment is roughly two days a month, and with such a level of commitment a person might spend one of those days at a project meeting in their own city and the other day working independently to prepare for the next meeting.

4 Project Organisation

This project unifies three existing projects, spread across six universities in three Australian states:

- **Exam Question Classification:** As described earlier, this sub-project is investigating the nature and composition of formal examination instruments used in summative assessment of introductory programming students, and the pedagogical intentions of the educators who construct these instruments.
- **Syllabus Specification:** This sub-project builds upon the CUSP system discussed earlier, to create a new system into which we can map the detailed curriculum of programming subjects.
- **Exam Question Generation and Benchmarking:** This sub-project aims to include some common questions in exams at participating institutions, and to benchmark student performance on those questions.

Each month there will be up to three one-day meetings in Brisbane, Sydney and/or Melbourne. The 'major' meeting will be attended by all participants from the city where it is held, along with one representative from each participating institution outside that city. One or two further meetings will be smaller, involving just the

project leader and the project participants located in that city.

4.1 Rules of Engagement

All project participants have agreed to the following rules about how they will work together:

- All members will help to procure the outlines and other public documents for the relevant programming subjects at their respective universities.
- All members will assist in rewriting those documents into the CUSP-derived system adopted by the project.
- Project members who teach one of the programming subjects will:
 - Provide other project members with the opportunity to run short, in-class, formative assessment exercises.
 - Give full consideration to using in their summative tests and exams the questions devised as part of this project. However, the final decision on the inclusion of any summative test or exam question remains with that teacher.
 - Consider contributing some of their own questions to a common, public pool of questions.
- All members will approach the (other) teachers of the first three programming subjects at their institutions, to request test papers, exam papers, and class performance data on those papers. Ideally, the papers thus solicited will be public domain, but the project will also accept and use papers on the understanding that they are to remain confidential.
- At least one member at each participating institution will complete any necessary ethical clearance process.
- Members interested in particular sub-projects will undertake tasks related to those sub-projects. Specifically:
 - Members most interested in syllabus specification will benchmark the enhanced CUSP system by encoding information about subjects at other institutions.
 - Members most interested in the exam question classification/archive will develop meta-tags for the classification/archive that has CUSP as its starting point, and will archive questions and information from the benchmarking of questions. They will also conduct and analyse interviews with relevant academics to explore the processes of writing and marking programming exams.
 - Members most interested in question benchmarking will provide their questions and student performance data for the classification/archive.
- In addition to the specific tasks listed above, all members will make some contribution across all parts of the project, and will actively pursue connections between the sub-project of primary interest to them and the other sub-projects.

Further matters still to be agreed by the participants include a protocol for authorship in papers produced within the project. What level of contribution is required

to warrant recognition as an author? In what order should the authors' names be listed? To these and similar questions Lister and Edwards (2010) propose answers that might well be adopted for the BABELnot project.

5 Evaluation Framework

A requirement of ALTC-funded projects of this size is that they have an external reference group and be formally evaluated by an outside evaluator. This is not an activity that we plan to leave until the end of the project. Instead, formative evaluation throughout will facilitate the attainment of better project outcomes.

While the most obvious role of the six-monthly workshops is dissemination, evaluation will also figure prominently at all the workshops. For example, by having workshop attendees perform training exercises such as classifying exam questions in the CUSP-derived formalism, we will collect evaluation data on whether the formalism can be understood easily and applied reliably.

The external evaluator will be appointed six months after the project begins, and will then work with the project team to develop an evaluation plan. The key sources of information for the evaluation will be: reports from the monthly meetings, dissemination events, and reference group review meetings; interviews with project members; and feedback sought from project members via surveys.

A working group will be proposed for the ITiCSE conference to be held in the UK in mid-2013. If the proposal is accepted, this working group will contribute to the summative evaluation of the project. An ITiCSE working group will provide a fresh set of academics, independent of the formative evaluations.

6 Conclusion

This paper represents a break in tradition, driven by our focus on dissemination. Traditionally, innovative education projects focus on their product, and do not report on their activities until either those activities are over, or at least a significant milestone has been attained. We describe that as the 'disseminate late' approach, and argue that this traditional approach has probably contributed to poor dissemination outcomes. Instead, we advocate a 'disseminate early and often' approach, which is what we are implementing in this project, primarily through the six-monthly workshops, but also through the very act of writing this paper – we are like software engineers who advocate writing the software tests early, even before writing the code, for well known reasons that we believe are analogous to why innovative education projects should begin dissemination early.

Through beginning the dissemination early, we now have the luxury of using this paper to invite others to participate in this project. Most may elect to attend any of the six monthly workshops, but others may accept our invitation to take on a more active role, and join us in the monthly meetings.

It is just over 10 years since McCracken et al's (2001) paper appeared. That paper directly and indirectly inspired a raft of multi-institutional collaborations in computing education. However, most of those projects were either short-lived, or (like BRACElet) were loosely organised, with much of the work being done without

formal financial support. Our project may be a prototype for new generation of multi-institutional projects. Our project is formally funded, and with that comes a commitment to deliver. That in turn leads to a more formally organised project structure. Also, applying lessons that have been learnt over the last ten years, the project has dissemination built explicitly into its structure.

As previously mentioned, this project builds upon the University of Sydney's existing CUSP system, which is currently used for over 240 degrees across four faculties at that institution. It is therefore possible that, after this project is formally over, the extended version of CUSP would be adopted by the current users: a cycle of innovation would be completed, by applying the outcomes and deliverables of this project to disciplines other than computing.

7 The Name BABELnot

Here is a brief explanation for readers who are curious about the name BABELnot. The name is deliberately reminiscent of the BRACElet project, of which Raymond Lister was a leader. The capitalisation of BRACElet reflects its continuation from the BRACE project, in which the name BRACE was an acronym. The name BABELnot calls to mind the biblical tale of the Tower of Babel, whose builders failed in their task because they lost the ability to communicate with one another. So while Babel highlights the challenge of communication when we speak different languages, BABELnot will help overcome that challenge by devising a common language in which programming educators may better communicate with one another on matters of standards and assessment within their subjects.

Acknowledgements

This project was approved for funding by the Australian Learning and Teaching Council, and is funded under the auspices of the Australian Federal Government's Department of Education, Employment and Workplace Relations (DEEWR). However, the views expressed in this paper are solely those of the authors, and not the views of the ALTC or DEEWR.

References

- ACM/IEEE (2008). Computer Science Curriculum 2008: An Interim Revision of CS 2001. <http://www.acm.org/education/curricula/ComputerScience2008.pdf>.
- Australian Learning and Teaching Council (2006). ALTC Dissemination Framework. <http://www.altc.edu.au/resource-dissemination-framework-altc-2008>
- Biggs, JB & KF Collis (1982). Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome). New York: Academic Press.
- Bloom, BS (1956). Taxonomy of Educational Objectives: Handbook I: Cognitive Domain. Longmans, Green and Company.
- Carter, J, J English, K Ala-Mutka, M Dick, W Fone, U Fuller, & J Sheard (2003). How shall we assess this? SIGCSE Bulletin 35(4):107-121.
- Clear, T, J Whalley, P Robbins, A Philpott, A Eckerdal, M-J Laakso, & R Lister (2011). Report on the final BRACElet workshop: Auckland University of Technology, September 2010. Journal of Applied Computing and Information Technology 15(1).
- Corney, M, D Teague, A Ahadi, & R Lister (2012). Some empirical results for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. 14th Australasian Computing Education Conference (ACE 2012), Melbourne, Australia.
- Gannaway, D, T Hinton, B Berry, & K Moore (2011). A review of the dissemination strategies used by projects funded by the ALTC Grants Scheme. Sydney: Australian Learning and Teaching Council.
- Gluga, R, J Kay, & T Lever (2010). Modeling long term learning of generic skills. Tenth International Conference on Intelligent Tutoring Systems, Pittsburgh, PA, USA, 85-94.
- Gluga, R, J Kay, R Lister, S Kleitman, & T Lever (2012). Coming to terms with Bloom: an online tutorial for teachers of programming fundamentals. 14th Australasian Computing Education Conference (ACE 2012), Melbourne, Australia.
- Lister, R (2011) Concrete and other neo-Piagetian forms of reasoning in the novice programmer. 13th Australasian Computing Education Conference, Perth, Australia, 9-18.
- Lister R, ES Adams, S Fitzgerald, W Fone, J Hamer, M Lindholm, R McCartney, JE Moström, K Sanders, O Seppälä, B Simon, & L Thomas (2004). A multi-national study of reading and tracing skills in novice programmers. SIGCSE Bulletin 36(4):119-150.
- Lister, R, T Clear, Simon, DJ Bouvier, P Carter, A Eckerdal, J Jacková, M Lopez, R McCartney, P Robbins, O Seppälä, & E Thompson (2010). Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. SIGCSE Bulletin 41(4):156-173.
- Lister, R & J Edwards (2010). Teaching novice computer programmers: bringing the scholarly approach to Australia – a report on the BRACElet project. Australian Learning and Teaching Council.
- Lister, R, C Fidge, & D Teague (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. 14th Conference on Innovation and Technology in Computer Science Education (ITiCSE 2009), Paris, France, 161-165.
- 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):125-140.
- McGettrick, A, R Boyle, R Ibbett, J Lloyd, L Lovegrove, & K Mander (2005). Grand challenges in computing: education – a summary. The Computer Journal 48(1):42-48.
- McKenzie, J, S Alexander, C Harper, & S Anderson (2005). Dissemination, adoption and adaptation of

- project innovations in higher education. Sydney: University of Technology, Sydney.
- Sheard, J, Simon, M Hamilton, & J Lönnberg (2009). Analysis of research into the teaching and learning of programming. Fifth International Workshop on Computing Education (ICER 2009), Berkeley, CA, USA, 93-104.
- Sheard, J, Simon, A Carbone, D Chinn, M-J Laakso, T Clear, M de Raadt, D D'Souza, J Harland, R Lister, A Philpott, & G Warburton (2011). Exploring programming assessment instruments: a classification scheme for examination questions. Seventh International Computing Education Research Workshop (ICER 2011), Providence, RI, USA, 33-38.
- Shuhidan, S, M Hamilton, & D D'Souza (2010). Instructor perspectives of multiple-choice questions in summative assessment for novice programmers. *Computer Science Education* 20:229-259.
- Simon, A Carbone, M de Raadt, R Lister, M Hamilton, & J Sheard (2008). Classifying computing education papers: process and results. Fourth International Workshop on Computing Education (ICER 2008), Sydney, NSW, Australia, 161-171.
- Simon, J Sheard, A Carbone, D Chinn, M-J Laakso, T Clear, M de Raadt, D D'Souza, R Lister, A Philpott, J Skene, & G Warburton (2012). Introductory programming: examining the exams. 14th Australasian Computing Education Conference (ACE 2012), Melbourne, Australia.
- Southwell, D, D Gannaway, J Orrell, D Chalmers, & C Abraham (2005). Strategies for effective dissemination of project outcomes. Carrick Institute for Learning and Teaching in Higher Education.
- Southwell, D, D Gannaway, J Orrell, D Chalmers, & C Abraham (2010). Strategies for effective dissemination of the outcomes of teaching and learning projects. *Journal of Higher Education Policy and Management* 32(1):55-67.
- Teague, D, M Corney, A Ahadi, & R Lister (2012). Swapping as the 'Hello World' of Relational Reasoning: Replications, Reflections and Extensions. 14th Australasian Computing Education Conference (ACE 2012), Melbourne, Australia.
- Whalley, J, R Lister, E Thompson, T Clear, P Robbins, PKA Kumar, & C Prasad (2006). An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. Eighth Australasian Computing Education Conference (ACE 2006), Hobart, Australia, 243-252.
- Wright, S, R Hadgraft, & I Cameron (2010). Learning and Teaching Academic Standards Project: Engineering and ICT. Australian Learning and Teaching Council.

Introductory programming: examining the exams

Simon

University of Newcastle
simon@newcastle.edu.au

Judy Sheard

Monash University
judy.sheard@monash.edu.au

Angela Carbone

Monash University
angela.carbone@monash.edu.au

Donald Chinn

University of Washington, Tacoma
dchinn@u.washington.edu

Mikko-Jussi Laakso

University of Turku
milaak@utu.fi

Tony Clear

Auckland University of Technology
tony.clear@aut.ac.nz

Michael de Raadt

Moodle
michaeld@moodle.com

Daryl D'Souza

RMIT University
daryl.dsouza@rmit.edu.au

Raymond Lister

University of Technology Sydney
raymond.lister@uts.edu.au

Anne Philpott

Auckland University of Technology
aphilpot@aut.ac.nz

James Skene

Auckland University of Technology
james.skene@aut.ac.nz

Geoff Warburton

Australia
geoffw173@gmail.com

Abstract

This paper describes a classification scheme that can be used to investigate the characteristics of introductory programming examinations. The scheme itself is described and its categories explained. We describe in detail the process of determining the level of agreement among classifiers, that is, the inter-rater reliability of the scheme, and we report the results of applying the classification scheme to 20 introductory programming examinations. We find that introductory programming examinations vary greatly in the coverage of topics, question styles, skill required to answer questions and the level of difficulty of questions. This study is part of a project that aims to investigate the nature and composition of formal examination instruments used in the summative assessment of introductory programming students, and the pedagogical intentions of the educators who construct these instruments.

Keywords: examination papers, computing education, introductory programming

1 Introduction

There are several common forms of assessment in introductory programming courses. In-class computer-based tests and programming assignments are good ways of assessing the interactive skill of designing and writing computer programs. Written quizzes and examinations are appropriate for assessing students' familiarity with relevant theoretical knowledge. In addition, written tests can examine some aspects of program designing and coding, although they are perhaps not ideally suited for the assessment of these skills.

Formal examinations are widely used in the summative assessment of students in programming courses. Writing an examination paper is an important task, as the exam is used both to measure the students' knowledge and skill at the end of the course and to grade and rank students. Yet it is often a highly individual task, guided by the whims, preferences, beliefs, and perhaps inertia of the examiner. Lister (2008) observes that there is a great deal of 'folk pedagogy' in computing education, and acknowledges that his early examinations were based upon folk-pedagogic misconceptions.

In constructing an exam, educators must consider what they wish to assess in terms of the course content. They must consider the expected standards of their course and decide upon the level of difficulty of the questions. Elliott Tew (2010) suggests that "the field of computing lacks valid and reliable assessment instruments for pedagogical or research purposes" (p.xiii). If she is right, and the instruments we are using are neither valid nor reliable, how can we make any credible use of the results?

An analysis of research papers about programming education published in computing education conferences from 2005 to 2008 found that 42% of the studies gathered data from formal exam assessment (Sheard, Simon, Hamilton & Lönnberg 2009). It seems critical that we understand the nature of these assessment instruments. Lister (2008) urges computing educators to base their decisions upon evidence. At least part of the relevant evidence should be an overview of introductory programming exams as a whole, and we have therefore set out to examine the examinations in introductory programming courses.

In this paper we describe an exam question classification scheme that can be used to determine the content and nature of introductory programming exams. We apply this instrument to a set of exam papers and describe the process of establishing a satisfactory inter-rater reliability for the classifications. We report what we have found about the content and nature of the exam papers under consideration. This study is the first step of large-scale investigation of the nature and composition of

formal examinations used in introductory programming courses, and the pedagogical intentions of the educators who write and use these examinations.

2 Background

Assessment is a critical component of our work as educators. Formative assessment is a valuable tool in helping students to understand what they have achieved and in guiding them to further achievement. Summative assessment is a tool used to determine and report the students' achievement, typically at the end of a course, and to rank the students who have completed the course. When we write and mark summative assessment instruments we are standing in judgment on our students. Yet concern has been expressed that very little work has gone into understanding the nature of the instruments that we use (Elliott Tew & Guzdial 2010).

A number of research studies have used examination instruments to measure levels of learning and to understand the process of learning. A body of work conducted under the auspices of the BRACElet project has analysed students' responses to examination questions (Clear et al 2008, Lister et al 2010, Lopez et al 2008, Sheard et al 2008, Venables et al 2009). Interest in this work stemmed from earlier studies, such as that of Whalley et al (2006), which attempted to classify responses to examination questions using Bloom's taxonomy (Anderson & Sosniak 1994) and the SOLO taxonomy (Dart & Boulton-Lewis 1998). The BRACElet project has focused on exam questions that concern code tracing, code explaining, and code writing. In an analysis of findings from these studies, Lister (2011) proposes that a neo-Piagetian perspective could prove useful in explaining the programming ability of students; this proposal could well guide future investigations into assessment in programming courses.

Few studies were found that investigated the characteristics of examination papers and the nature of exam questions. A cross-institutional comparative study of four mechanics exams by Goldfinch et al (2008) investigated the range of topics covered and the perceived level of difficulty of exam questions. Within the computing discipline, Simon et al (2010) analysed 76 CS2 examination papers, but considered only particular data structures questions, which made up less than 20% of the marks available in the exams. Their analysis focused on the cognitive skills required and the level of difficulty of the questions. Following this study, a further analysis of 59 CS2 papers in the same dataset (Morrison et al 2011) investigated the range of question styles that can be used to test students' skills in the application of data structures. Petersen et al (2011) analysed 15 CS1 exam papers to determine the concepts and skills covered. They found a high emphasis on code writing questions, but with much variation across the exams in the study. Shuhidan et al (2010) investigated the use of multiple-choice questions in summative assessment of four levels of programming courses (CS0-CS3) and found that the use of these questions remained controversial.

Our study focuses on introductory programming examination papers, developing a classification scheme for the purpose of analysing these papers to give a

comprehensive view of the style of questions that make up these instruments.

The study was initiated at a workshop associated with the Fourth International Workshop on Computing Education (ICER 2008). A small group developed the ideas and a provisional classification scheme, which they presented at a subsequent workshop associated with the 13th Australasian Computing Education Conference (ACE 2011). At the second workshop the scheme was trialled on a few exam questions and adjusted in the light of the trial. Full details of the scheme's development are described elsewhere (Sheard et al 2011).

3 The classification scheme

The classification scheme consists of eleven different measures, one of which is administrative and the other ten of which describe features that we believe are useful in trying to form an understanding of an examination.

The remainder of this section briefly describes each of the features, and, where appropriate, the reasons for their inclusion.

Percentage of mark allocated. This is the feature described above as administrative. While it might be inherently useful, for example in noting whether comparable questions are worth comparable marks in different exams, its principal purpose in this scheme is for weighting, determining what proportion of a complete exam covers the mastery of particular topics or skills.

Topics covered. In the classification system used here an exam question is assigned at most three of the following topics: data types & variables, constants, strings, I/O, file I/O, GUI design and implementation, error handling, program design, programming standards, testing, scope (includes visibility), lifetime, OO concepts (includes constructors, classes, objects, polymorphism, object identity, information hiding, encapsulation), assignment, arithmetic operators, relational operators, logical operators, selection, loops, recursion, arrays, collections (other than arrays), methods (includes functions, parameters, procedures and subroutines), parameter passing, operator overloading.

In the list above, topics that follow 'assignment' tend to subsume data types & variables, so any question that is categorised with these later topics need not include data types & variables. Similarly, a topic such as selection or loops usually subsumes operators, and arrays generally subsumes loops. Having assigned one of these broader topics to a question, we would not also assign a topic subsumed by that broader topic.

The list of topics was compiled from a number of different sources, including the computing education literature. Dale (2005, 2006) lists the topics that emerged from a survey of computing academics; Schulte and Bennedsen (2006) surveyed teachers of introductory programming to determine the topics that were taught and the perceived difficulty of those topics; Elliott Tew and Guzdial (2010) identified topics by analysing the content of relevant textbooks.

Skill required to answer the question. Some questions can be answered purely by recalling knowledge that has been imparted during the course. Others require the application of different skills: tracing code (which

includes evaluating expressions), explaining code, writing code, modifying code (which includes refactoring or rewriting code), debugging code, designing programs, and testing programs. When classifying a question we require a single skill to be nominated. If a question appears to require two or more skills (for example, designing programs and writing code), we would classify it with the skill that appears dominant: in a question involving program design and code-writing, the code-writing would probably dominate.

Style of question. This feature indicates what form of answer is expected by the question. The choices are: multiple choice, short answer (including definitions, results of tracing or debugging, and tables), program code, Parsons problem (Parsons & Haden 2006), and graphical representation (for example, concept, flow chart, class diagram, picture of a data structure). Only one of the above can be chosen. Similar categories were used by Petersen et al (2011).

Open/closed. A question that has only one possible correct answer is classified as closed. All others are classified as open.

Cultural references. Is there any use of terms, activities, or scenarios that may be specific to a cultural group and may influence the ability of those outside the group to answer the question? There might be references to a particular ethnic group and their customs, but a cultural reference need not be ethnic. For example, a question might use vocabulary or concepts that refer to a specific sport, such as cricket.

Degree of difficulty. Low, medium, or high. This is an attempt to estimate how difficult the average student would find the question at the end of an introductory course. This classification is similar to that used by Simon et al (2010) in their analysis of CS2 exam papers and Goldfinch et al (2008) in their analysis of mechanics examination papers.

For reasons explained in the next section, the remaining five measures were not used in the current analysis, and therefore their description here is far more brief than the description given to and used by the classifiers.

Explicitness. Low, medium, or high. Extent to which the question states explicitly what the students need to know in order to answer the question. A question with low explicitness will assume that students already know, or can deduce, much about the task to be completed.

Operational complexity. Low, medium, or high. The number and sophistication of the tasks to be performed.

Conceptual complexity. Low, medium, or high. The types and combinations of the concepts that must be known in order to correctly answer the question.

Linguistic complexity. Low, medium, or high. The length, sophistication, and general comprehensibility of the question.

Intellectual complexity. Where the question fits into Bloom's taxonomy (Anderson & Sosniak 1994).

The measures of complexity were originally used by Williams and Clarke (1997) in the domain of

mathematics, and were applied to the computing domain by Carbone (2007).

4 Inter-rater reliability

As mentioned in Section 2, a number of studies have classified examinations in various ways. However, none of those studies has really established whether their classification systems are reliable across multiple raters.

Simon et al (2010) report at least 80% agreement on their classification, but in each instance this was between just two classifiers, one of whom classified a full set of questions and the other of whom classified 20% of those questions to check the level of agreement. Furthermore, their analysis deals only with questions in highly specific topics, and the questions classified by each main classifier were all in the same topic area. It is conceivable that all of these factors would contribute to a higher level of agreement than might be expected among a large number of classifiers analysing a broader range of questions.

Petersen et al (2011) did not conduct an inter-rater reliability test. After classifying the questions they were considering, the individual classifiers discussed their classifications in an attempt to reach consensus. Even then, they report difficulty in reaching consensus on most of the measures they were applying.

Goldfinch et al (2008) do not report an attempt to measure agreement among the classifiers. Like Petersen et al (2011) they classified individually and then attempted to reach consensus, and like Petersen et al they found it remarkably difficult to do so.

For this project we chose to conduct a formal and transparent test of inter-rater reliability. With few such tests reported in the computing education literature, we felt it important to conduct and to report on this test.

4.1 Reliability test 1: individual

The first test of inter-rater reliability was carried out on the full scheme of 11 categories. All 12 participants independently classified the 33 questions of the same examination in all 11 categories.

All categories but one were analysed using the Fleiss-Davies kappa for inter-rater reliability (Davies & Fleiss 1982). Because the scheme permits multiple topics to be recorded for a question, the Topics category could not be analysed by this measure, which depends upon the selection of single values.

Table 1 shows the results of the inter-rater reliability test. On kappa measurements of this sort, an agreement of less than 40% is generally considered to be poor; between 40% and 75% is considered fair to good; and more than 75% is rated excellent (Banerjee et al 1999).

Perhaps the most startling figure in Table 1 is the 73% agreement on the percentage mark for each question. This was simply a matter of copying the mark for each question from the exam paper to the spreadsheet used for classifying. The bulk of the disagreement was due to one classifier who neglected to enter any values for the percentage mark. Once this was remedied, the agreement was still only 98%, because two classifiers had each wrongly copied one value. This is a salutary reminder that data entry errors do happen, and we resolved that all of

Category	Reliability	Reliability range
Percentage	73%	fair to good
Skill required	73%	fair to good
Style of question	90%	excellent
Open/closed	60%	fair to good
Cultural references	15%	poor
Degree of difficulty	43%	fair to good
*Explicitness	31%	poor
*Operational complexity	52%	fair to good
*Conceptual complexity	34%	poor
*Linguistic complexity	47%	fair to good
*Intellectual complexity	27%	poor

Table 1: Inter-rater reliability for 11 categories of the initial scheme (the 12th cannot be analysed by this measure). *The categories marked with asterisks were dropped for the classifying reported in this paper.*

our subsequent classifying would be conducted by pairs, in the expectation that this would help to eliminate such errors.

While we did not expect full agreement on the other measures, we were still surprised at the extent of disagreement. More often than not, each classifier felt reasonably confident that they could at least determine how difficult a question is; yet agreement on that measure was an uninspiring 43%. Like Goldfinch et al (2008) and Petersen et al (2011) we realised that the difficulty of a question is strongly dependent on what was taught in the course and how it was taught, and without that information we could only rate the questions according to how difficult we believed our own students would find them.

Following this rather disappointing result, the five categories dealing with complexity, marked in Table 1 with asterisks, were dropped from the scheme until we could find a way to improve the reliability of classification on those measures.

In view of its exceedingly poor agreement, it might seem strange that we did not drop the cultural references category at this point. One reason is that the nature of the disagreements was different. On the complexity measures the classifications tended to be spread more or less evenly across the possible values, and we hope that with further clarification and practice it will be possible to reduce the spread. On the cultural references measure the disagreement was invariably that one classifier saw a reference that others had not seen, but tended to acknowledge after discussion. This is discussed further in section 5.6.

4.2 Reliability test 2: individual

Having thoroughly classified one exam paper in the first inter-rater reliability test, we classified a further ten exams according to the remaining categories. Classifiers worked in pairs, first classifying each question individually, then discussing their classifications and seeking consensus where there was disagreement.

Category	Fleiss-Davies Kappa		
	Test 1 (solo)	Test 2 (solo)	Test 3 (pair)
Percentage	73%	100%	100%
Skill required	73%	73%	84%
Style of question	90%	89%	93%
Open/closed	60%	73%	86%
Cultural references	15%	33%	37%
Degree of difficulty	43%	54%	60%

Table 2: Inter-rater reliability for six categories of the interim scheme (the seventh cannot be analysed by this measure)

A second inter-rater reliability test was then conducted to determine whether the additional practice and the experience of working in pairs had improved the level of agreement. Again all 12 participants classified a single complete examination, this one consisting of 28 questions. For completeness, it should be noted that at this point one of the original 12 members became unavailable to continue with the work, and a new member joined the project.

4.3 Reliability test 3: pairs

When two classifiers disagree, this is either because one of them has made a minor error, which should be picked up immediately, or because there is genuine scope for disagreement. In the latter case, two people discussing the question might be more likely than one person alone to reach the same conclusion as others. For this reason, immediately following the second inter-reliability test the individual classifiers were formed into pairs and asked to agree on each of the classifications of that same examination.

The pairs for this third test were not self-selected, and were generally not the same as the pairs that had worked together on the first set of classifications. Instead they were selected by their order of completion of the individual reliability test. When the first two classifiers had completed their individual classification of the exam questions, they were formed into a pair and asked to come up with an agreed classification for the same questions; when the next two individuals had finished, they were formed into the second pair; and so on.

4.4 Comparing the reliability tests

Table 2 shows the results of all three inter-rater reliability tests on the six categories that they have in common.

It is pleasing to see that between the first two tests, reliability generally improved with time and practice.

It is also pleasing to see that the agreement between pairs in the third test was an improvement on the agreement between individuals in the second test. On the basis of this finding, we conclude that pair classification is more reliable than individual classification.

Neither of these findings is surprising, but such findings are seldom reported, so we feel that there is value in explicitly reporting them here. On the basis of the second finding, we plan to conduct all of our subsequent classification in pairs.

5 Results

This section presents the results of analysing 20 introductory programming exam papers using the exam classification scheme. A total of 469 questions were identified in these exams, with the number of questions in an exam ranging from 4 to 41. For each question the percentage mark allocated was recorded, and this was used as a weighting factor when calculating the contribution of each question to the values in each category.

5.1 Exam paper demographics

The 20 exam papers in the study were sourced from ten institutions in five countries. They were all used in introductory programming courses, eighteen at the undergraduate level and two at the postgraduate level. Course demographics varied from 25 students on a single campus to 800 students over four domestic and two overseas campuses. Most courses used Java with a variety

of IDEs (BlueJ, JCreator, Netbeans, Eclipse), one used JavaScript, one used C# with Visual Studio, one used Visual Basic, one used VBA (Visual Basic for Applications) and one used Python. Table 3 shows further specific information about the 20 papers and the courses in which they are used.

5.2 Topics covered

For each question we recorded up to three topics that we considered were central to the question. From our original set of 26 topics, two topics (algorithm complexity and operator overloading) did not appear in the data set; and during analysis we added four further topics (events, expressions, notional machine and class libraries), giving a final list of 28 topics.

Table 4 shows the topics classified and their percentage coverage over the exams in the sample. Topics with the greatest coverage were OO concepts, methods, loops, arrays, program design, I/O and

Paper	Paper source	Exam characteristics				Teaching context			
	Country	Format	Style	% of final mark	Duration (hrs)	Enrolment	Mode	Approach	Program ming language
1	New Zealand	Paper	Closed book	40	2	150-200	Campus	Objects first	Java
2	New Zealand	Paper	Closed book	40	2	180	Campus	Objects first	Java (Karel the robot)
3	Australia	Paper	Closed book	40	3	240	Campus	Objects first	Java
4	Australia	Paper	Closed book	50	2	450	Online	Programming logic, then Java	Alice, Java
5	Australia	Paper	Closed book	50	3	120	Campus	Objects later	Java
6	Australia	Paper	Closed book	50	3	250	Campus	Objects later	Visual Basic
7	Australia	Paper	Closed book	50	3	50	Mixed	Objects first	Java
8	Australia	Paper	Closed book	50	3	255	Campus	Objects later	C#
9	Australia	Paper	Closed book	60	2	250	Campus	Objects first	Java
10	Australia	Paper	Closed book	60	2.5	60	Campus	Objects later	VBA
11	Australia	Paper	Closed book	60	3	700-800	Campus	Objects later	Java
12	Australia	Paper	Closed book	60	3	700-800	Mixed	Objects later	Java
13	Australia	Paper	Closed book	60	3	700-800	Mixed	Objects later	Java
14	Finland	Paper	Closed book	70	3	20	Campus	Procedural	Python
15	Finland	Paper	Closed book	80	3	60	Mixed	Objects later	Java
16	England	Paper	Closed book	80	3	100	Campus	Objects later	Java
17	Australia	Paper	Mixed	50	2	180	Mixed	Web script, procedural	JavaScript
18	Australia	Paper	Mixed	70	2	337	Campus	Objects first	Java
19	USA	Paper & online	Open book	25	2	25	Campus	Objects later	Java
20	USA	Paper & online	Open book	25	4	30	Campus	Objects later	Java

Table 3: Exam papers classified in this study

Topic	% Coverage
OO concepts (includes constructors, classes, objects, polymorphism, object identity, information hiding, encapsulation)	35.8
Methods (includes functions, parameters, procedures and subroutines)	34.5
Loops (subsumes operators)	32.3
Arrays	26.3
Program design	16.9
I/O	12.3
Selection (subsumes operators)	11.3
Assignment	8.2
File I/O	6.8
Parameter passing	6.7
Strings	6.2
data types& variables	4.4
Arithmetic operators	3.5
Error handling	3.1
Collections (other than arrays)	2.8
Relational operators	1.9
Scope (includes visibility)	1.8
GUI	1.8
Testing	1.3
Constants, Events, Expressions, Lifetime, Logical operators, Programming standards, Recursion	< 1 each

Table 4: Topics and their coverage over the 20 exams

selection. Eleven topics had less than 2% coverage.

The study conducted by Elliott Tew and Guzdial (2010) identified a set of eight concepts most commonly covered in CS1 courses; six of these appear in the top seven topics listed in Table 4. Their top eight concepts did not include program design, but included recursion and logical operators, which we found had low coverage (0.7% and 0.9% respectively).

5.3 Skill required

From a list of eight skills, each question was classified according to the main skill required to answer the question. Figure 1 shows the overall percentage coverage of each required skill over the 20 exams in the dataset. The most frequently required skill was code writing (48%). The five skills concerning code (writing, tracing, explaining, debugging and modifying) together covered 81% of all exams, the remainder being taken by knowledge recall (10%), design (7%) and (2%) testing. We recognise that writing code often also involves a degree of program design, but we classified questions under program design only if they did not involve coding.

Figure 3 shows a summary of the skills required in each exam. In this graph the five skills associated with coding have been combined into a single coding category. The graph shows that coding in these various forms is the predominant skill assessed in introductory programming exams.

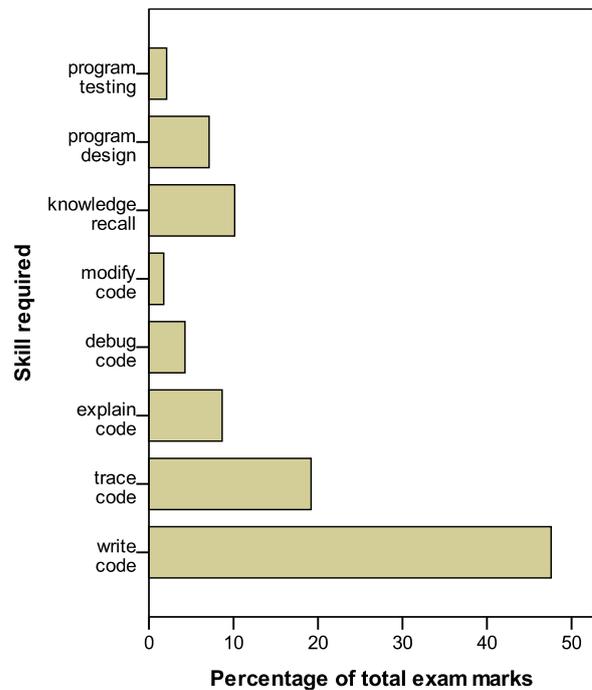


Figure 1: Skills required to answer questions

The four exams that exceed 100% do so because they include some choice, so students do not have to complete all questions to score 100%. The one exam that falls below 100% does so because it includes material other than programming, and we analysed only the programming-related questions.

5.4 Question style

The most common question style involved producing code, with 54% of the marks allocated for code-writing questions (including Parson's problems). Short-answer questions make up 28% of the exams, multiple-choice questions 17%, and graphical style less than 2% (see Figure 2). These findings are somewhat comparable with those of Petersen et al (2011), whose study of CS1 exam content found that 59% of the exam weight was for coding questions, 36% for short answer, 7% for multiple choice, and 3% for graphical questions.

Figure 4 summarises the question styles in each exam, and shows a wide variation across the exams. One exam consists entirely of multiple choice questions, while more than half have no multiple choice questions. It is interesting to note that although coding is the predominant style overall, in two exams there is no code writing required. Petersen et al (2011) also found that the percentage of code writing varied across the CS1 exams they studied.

5.5 Open/closed

The questions were coded according to whether they were open or closed in nature. More marks were allocated to questions that were open (61%) than closed (39%), but this varied markedly over the exams in our sample, as shown in Figure 5. In two exams all questions were closed, and in one exam all questions were open.

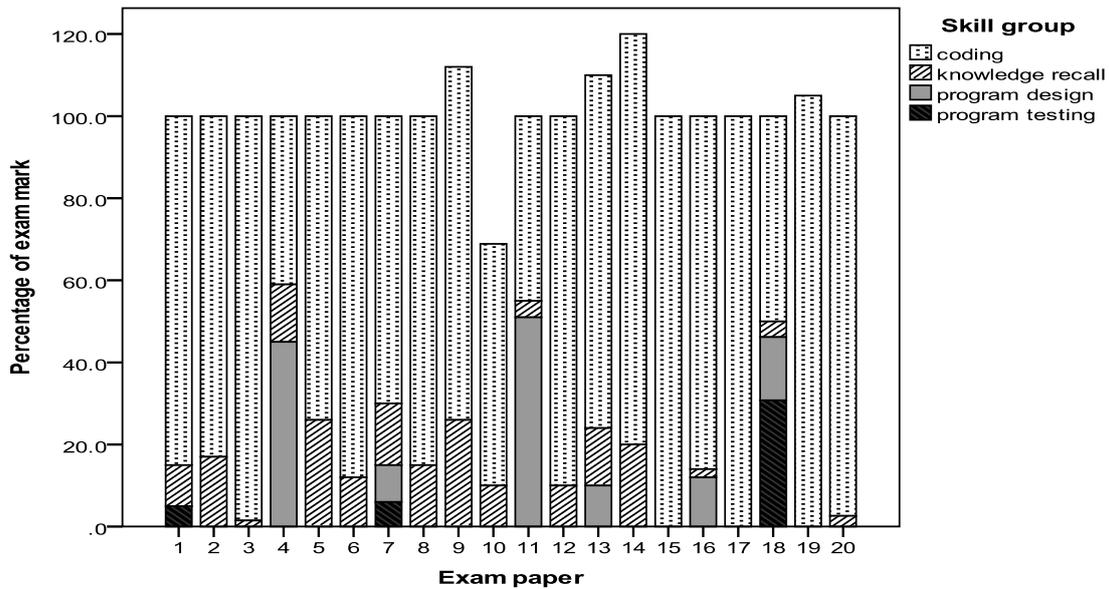


Figure 3: Skills required in each exam

5.6 Cultural references

Cultural references were identified in only eight of the 469 questions analysed, making up a little more than 2% of the available marks. This is so small as to suggest that it might not be worth assessing or reporting – especially as the trial classification showed that any cultural references tended to be spotted first by a single classifier, and only then agreed to by others. However, one likely extension of this work will be to establish a repository of exam questions for the use of educators. In such a repository, this category would serve to alert users that somebody feels a particular question may cause problems for some students outside a particular context or culture.

5.7 Level of difficulty

The questions were classified according to the perceived level of difficulty for a student at the end of an introductory programming course. Overall, half of the marks (50%) were allocated to questions rated as medium

difficulty, while low difficulty (26%) and high difficulty (24%) scored about the same. As with other categories, levels of difficulty varied greatly over the exams in our sample, as shown in Figure 6. By comparison, in the data structures questions that they analysed, Simon et al (2010) classified more questions as high (42%), fewer questions as medium (40%), and about the same proportion as low in difficulty.

6 Discussion

A number of computing education research groups are undertaking classification of various kinds, presumably sharing our belief that being able to accurately describe a concept is an important step on the road to understanding it. However, there is little point to a classification system unless it can be clearly established that the system is reliable across multiple classifiers.

In this paper we lay out the steps that were taken to assess the reliability of our scheme for classifying exam questions. We explicitly apply a recognised inter-rater reliability measure, developed and verified by statisticians, and we explain at which stages of the classification we applied this measure. We explain our decision to drop several categories of our scheme until we can find a way to improve the inter-rater reliability of those categories.

We are therefore able to provide evidence that reliability appears to improve as the classifiers do more classifying, and that classifying in pairs is more reliable than classifying by individuals.

We believe that an approach of this rigour is essential if readers are to have faith in the findings that we report.

The exams that we have analysed show a very heavy emphasis on coding skills, and the topics covered are concerned mainly with programming elements and constructs. This is not surprising in courses that teach programming, but it is worth noting that, while there was some coverage of the related topic of program design, there was very little focus on examining programming standards and program testing.

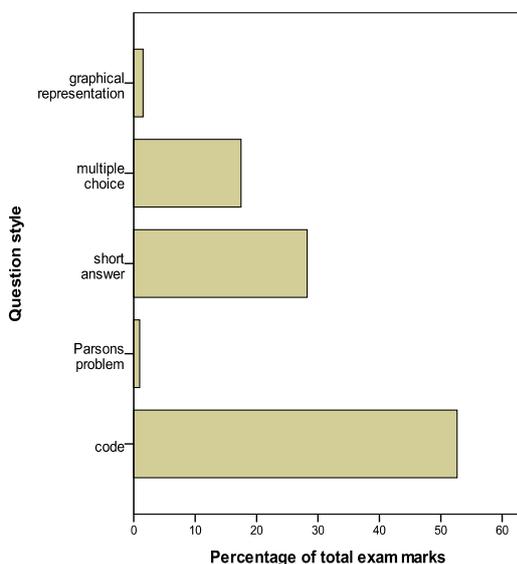


Figure 2: Marks awarded for each style of question

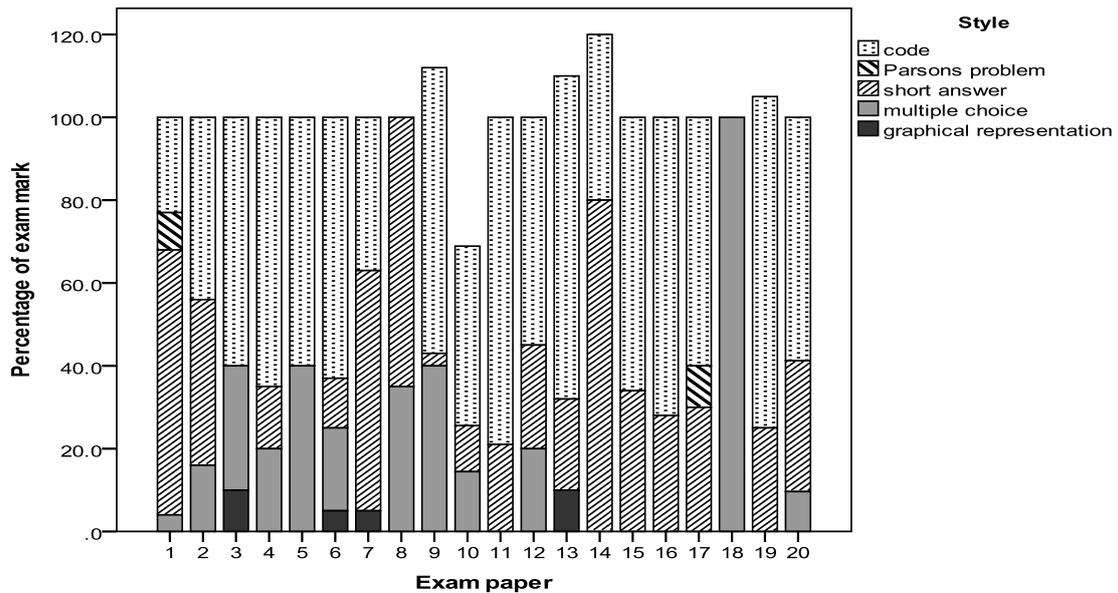


Figure 4: Marks for styles of question in each exam

Of course the skills being examined might not represent the full extent of what is taught in the course. Further material might be covered by assessment items other than the final exam, or indeed might not be assessed at all, even though it is explicitly taught. It is worth noting that a recent study by Shuhidan et al (2010) found that half the instructors surveyed believed that summative assessment was a valid measure of a student’s ability to program.

We have found a wide variation in the styles of question used in exams. Within a single exam, this variety could offer students a number of different ways to demonstrate what they have learned. Between exams it raises the question of whether different forms of questions are equally valid assessments of acquired programming skills. For example, more than half of the exams we analysed had a multiple choice component, and one of them was entirely multiple choice. The study by

Shuhidan et al (2010) found that the use of multiple choice questions is controversial. At this stage of our study we have not tried to determine why particular styles of question were used; we intend to pursue this question in our future work.

The variation among raters in the trial raises some interesting questions. Most of the participants are or have been involved in teaching introductory programming courses, yet the agreement on level of difficulty was only 43% in the first trial and 54% in the second, rising to 60% for the pair classification. Essentially, there was little or no consensus on whether questions were easy, moderate, or difficult. Both at the workshop and following the trial, discussion of specific questions brought out good arguments for each possible level of difficulty, making it clear that what we are trying to determine is highly subjective, and depends not just upon the feelings of individual participants but on their knowledge of the

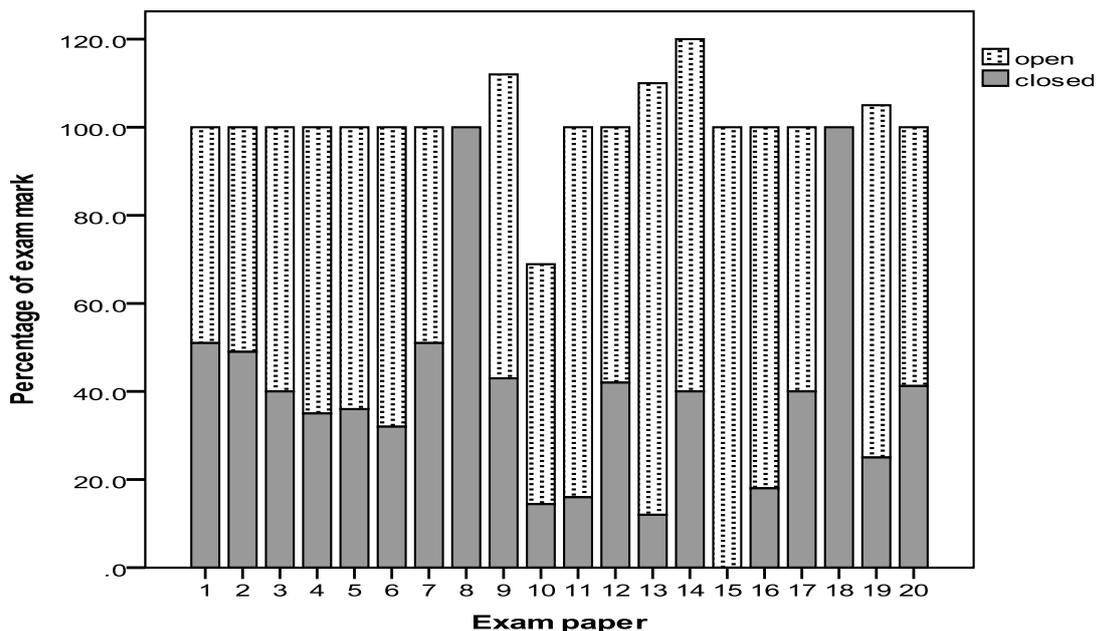


Figure 5: Marks for open and closed questions for each exam

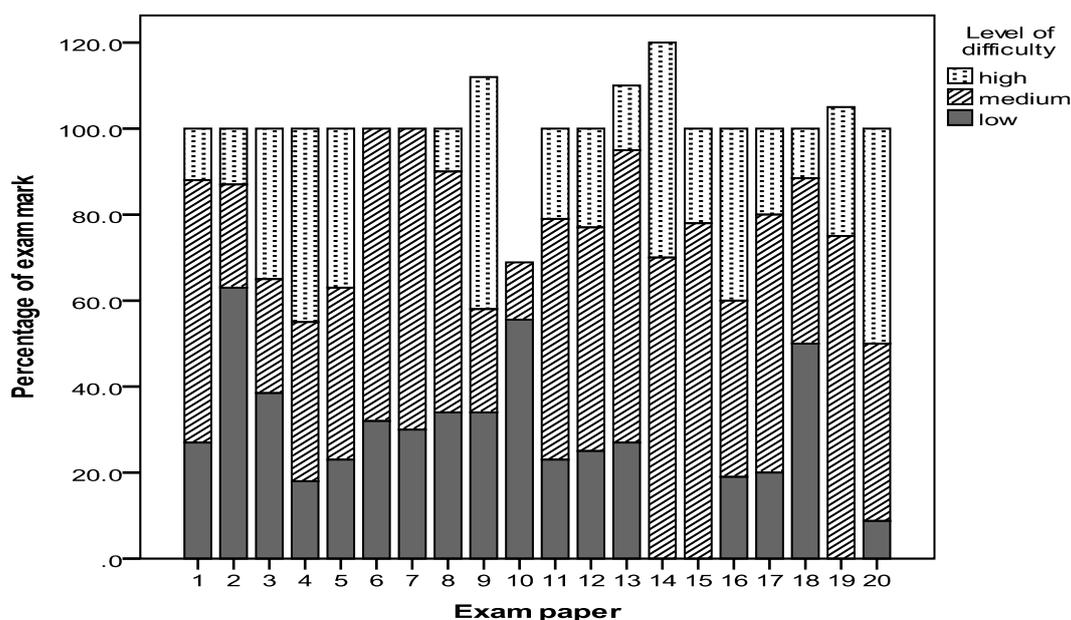


Figure 6: Level of difficulty for questions in each exam

courses that they teach and how their students would therefore respond to those particular questions. Perhaps it is also influenced in some small way by aspects of the culture of the institutions at which the individual participants are employed.

It is a consequence of this line of thought that, even with the appropriate training, the author of an exam is unlikely to classify it as we do except on the trivial measures. The author is fully conversant with the course and its context, and is thus better able to classify the exam within that context. Our classification, on the other hand, is being conducted in the context of the other introductory programming exams that we classify, with no detailed knowledge of how each individual course was taught. We are exploring the range of exams and exam questions that we encounter, from what we hope is a reasonably consistent perspective.

7 Future work

We have classified 20 introductory programming examinations, but this is not yet a large enough set to furnish a general picture of examinations in introductory programming courses. For example, all of these exams are in procedural and/or object-oriented programming. We plan to classify a broader set of examinations, including some from functional programming courses and some from logic programming courses. With this expanded data set we hope to be able to form a broad view of what introductory programming exams consist of.

In parallel with this further classification we intend to explore the role of formal examinations in programming courses. It is not obvious that a written examination of short duration is the best way to assess students' acquisition of a skill that is arguably best applied while working for longer periods at a computer. Why, then, do so many programming courses include a written exam as a major component of their assessment? We intend to interview a number of teachers of introductory programming courses in the hope of eliciting an answer to this question.

In addition, we hope that the interviews will give us an insight into how academics design and create their exams, and to what extent that process is tied in with the stated learning objectives of the course.

Once we have completed the additional classification and the interviews, we hope to be able to present a rich picture of the nature and role of examinations in introductory programming courses.

8 References

- Anderson, LW & LA Sosniak (1994). Excerpts from "Taxonomy of Educational Objectives, The Classification of Educational Goals, Handbook I: Cognitive Domain". In Bloom's Taxonomy: A Forty Year Retrospective, LW Anderson and LA Sosniak, Eds. Chicago, Illinois, USA: The University of Chicago Press, 9-27.
- Banerjee, M, M Capozzoli, L McSweeney, & D Sinha (1999). Beyond kappa: a review of interrater agreement measures, *Canadian Journal of Statistics* 27:3-23.
- Carbone, A (2007). Principles for designing programming tasks: how task characteristics influence student learning of programming. PhD dissertation, Monash University, Australia.
- Clear, T, J Whalley, R Lister, A Carbone, M Hu, J Sheard, B Simon, & E Thompson (2008). Reliably classifying novice programmer exam response using the SOLO taxonomy. 21st Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2008), Auckland, New Zealand, 23-30.
- Dale, N (2005). Content and emphasis in CS1. *SIGCSE Bulletin* 37:69-73.
- Dale, N (2006). Most difficult topics in CS1: Results of an online survey of educators. *SIGCSE Bulletin* 38:49-53.
- Dart, B & G Boulton-Lewis (1998). The SOLO model: Addressing fundamental measurement issues. *Teaching and Learning in Higher Education*, M. Turpin, Ed. Camberwell, Victoria, Australia: ACER Press, 145-176.

- Davies, M & JL Fleiss (1982). Measuring agreement for multinomial data, *Biometrics* 38:1047-1051.
- Elliott Tew, A (2010). Assessing fundamental introductory computing concept knowledge in a language independent manner. PhD dissertation, Georgia Institute of Technology, USA.
- Elliott Tew, A & M Guzdial (2010). Developing a validated assessment of fundamental CS1 concepts. SIGCSE 2010, Milwaukee, Wisconsin, USA, 97-101.
- Goldfinch, T, AL Carew, A Gardner, A Henderson, T McCarthy, & G Thomas (2008). Cross-institutional comparison of mechanics examinations: a guide for the curious. Conference of the Australasian Association for Engineering Education, Yeppoon, Australia, 1-8.
- Lister, R (2008). After the gold rush: toward sustainable scholarship in computing. Tenth Australasian Computing Education Conference (ACE 2008), Wollongong, Australia, 3-17.
- Lister, R (2011). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. 13th Australasian Computing Education Conference (ACE 2011), Perth, Australia, 9-18.
- Lister, R, T Clear, Simon, DJ Bouvier, P Carter, A Eckerdal, J Jacková, M Lopez, R McCartney, P Robbins, O Seppälä, & E Thompson (2010). Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *SIGCSE Bulletin* 41:156-173.
- Lopez, M, J Whalley, P Robbins, & R Lister (2008). Relationships between reading, tracing and writing skills in introductory programming. Fourth International Computing Education Research Workshop (ICER 2008), Sydney, Australia, 101-112.
- Morrison, B, M Clancy, R McCartney, B Richards, & K Sanders (2011). Applying data structures in exams. *SIGCSE 2011*, Dallas, Texas, USA, 631-636.
- Parsons, D & P Haden (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. Eighth Australasian Computing Education Conference (ACE 2006), Hobart, Australia, 157-163.
- Petersen, A, M Craig, & D Zingaro (2011). Reviewing CS1 exam question content. *SIGCSE 2011*, Dallas, Texas, USA, 631-636.
- Schulte, C & J Bennedsen (2006). What do teachers teach in introductory programming? Second International Computing Education Research Workshop (ICER 2006), Canterbury, UK, 17-28.
- Sheard, J, A Carbone, R Lister, B Simon., E Thompson, & J Whalley (2008). Going SOLO to assess novice programmers. 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2008), Madrid, Spain, 209-213.
- Sheard, J, Simon, M Hamilton, & J Lönnberg (2009). Analysis of research into the teaching and learning of programming. Fifth International Computing Education Research Workshop (ICER 2009), Berkeley, CA, USA, 93-104.
- Sheard, J, Simon, A Carbone, D Chinn, M-J Laakso, T Clear, M de Raadt, D D'Souza, J Harland, R Lister, A Philpott, & G Warburton (2011). Exploring programming assessment instruments: a classification scheme for examination questions. Seventh International Computing Education Research Workshop (ICER 2011), Providence, RI, USA, 33-38.
- Shuhidan, S, M Hamilton, & D D'Souza (2010). Instructor perspectives of multiple-choice questions in summative assessment for novice programmers, *Computer Science Education* 20:229-259.
- Simon, A Carbone, M de Raadt, R Lister, M Hamilton, & J Sheard (2008). Classifying computing education papers: process and results. Fourth International Computing Education Research Workshop (ICER 2008), Sydney, NSW, Australia, 161-171.
- Simon, B, M Clancy, R McCartney, B Morrison, B Richards, & K Sanders (2010). Making sense of data structures exams. Sixth International Computing Education Research Workshop (ICER 2010), Aarhus, Denmark, 97-105.
- Venables, A, G Tan, & R Lister (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. Fifth International Computing Education Research Workshop (ICER 2009), Berkeley, CA, USA, 117-128.
- Whalley, J, R Lister, E Thompson, T Clear, P Robbins, PKA Kumar, & C Prasad (2006). An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies, Eighth Australasian Computing Education Conference (ACE 2006), Hobart, Australia, 243-252.
- Williams, D & D Clarke (1997). Mathematical task complexity and task selection. Mathematical Association of Victoria 34th Annual Conference, Clayton, Vic, Australia, 406-415.

Student Created Cheat-Sheets in Examinations: Impact on Student Outcomes

Michael de Raadt

Moodle

michaeld@moodle.com

Abstract

Examinations have traditionally been classified as “open-book” or “closed-book” in relation to the freedom for students to bring resources into examinations. Open-book examinations can have benefits, such as reduced anxiety, de-emphasis of memorisation and reduced cheating. But open-book examinations can also have disadvantages such as reduced preparation and the need for time during examinations to look up facts. An emerging alternative allows students to bring a ‘cheat-sheet’ of hand-written notes. This form of examination has the potential to offer many of the benefits of an open-book examination while overcoming some of its failings. There has been little evidence showing that cheat-sheets can have an impact, and what exists is contradictory. This study reveals that students who create and use cheat-sheets performed better, on average, in an introductory programming examination. Certain features of cheat-sheets were found to be related to superior performance, which may relate to student understanding.

Keywords: cheat-sheet, examination, open-book, computing education, introductory programming

1 Introduction

Resources available to students can be controlled in order to allow students access to information during an examination. In a closed-book examination, students are not permitted to bring any information into the examination that may assist them; they are required to rely on their memory to recall the information they need. In the 1950s educationalists began to explore new forms of examination and one form suggested was the open-book examination. “In such an examination the student is allowed to make use of any materials at his disposal, including textbooks, lecture notes and dictionaries, but does not obtain answers directly or indirectly from other students” (Kalish, 1958, p. 200). Between these extremes, examiners can constrain students’ access to materials to varying degrees; this form of examination can be referred to as a restricted examination. One form of restricted examination allows students to access a teacher created set of notes that summarise facts needed during the examination. Another form of restricted examination allows students to bring their own prepared notes, or ‘cheat-sheets’, into the examination setting. The purpose

of this study is to explore the potential benefits of student created cheat-sheets within the context of an introductory programming examination.

This paper will first present an overview of research in the area of examinations relative to students’ freedom to bring materials. This will be followed by the description of a study of the use of cheat-sheets in an introductory programming course and the results of that study. Finally, conclusions and recommendations will be made.

1.1 Open Book Examinations

The use of textbooks and teacher-prepared notes has been a topic of study for some time, with published discussions dating back over 60 years. Tussing (1951) suggested that the advantages of open-book examinations can include:

- reduced anxiety in the form of “fear and emotional blocks”;
- a shift in emphasis from memorisation to reasoning; and
- reduction of cheating.

Experimental studies have reported that open-book examinations can have benefits. Schumacher et al. (1978) ran a controlled experiment to compare performance on open-book and closed-book examinations. They found significantly higher average scores when students had access to a textbook during their examination. Theophilides and Koutselini (2000) found evidence that student attitudes towards a course improve where there is an open-book examination.

There are also counter-arguments to the use of open-book examinations. These include a degradation of the seriousness of examinations that can lead to superficial learning (Kalish, 1958). Rather than benefitting students, allowing students to have access to textbooks or teacher-prepared notes can be a hindrance. Boniface (1985) compared the time spent referring to open-book materials and their results. He found that students who spend more time than others referring to such materials tend to end up with poorer marks. It tends to be students who have performed poorly on prior assessment items that rely more on these materials during an examination.

In an experimental evaluation of open book examinations, Kalish (1958) found that average scores were not affected when comparing open-book and closed-book examinations, and concluded that open-book examinations may benefit some students more than others. These findings were echoed by Bacon (1969). According to Feldhusen (1961), students prepare less for an open-book examination, which may ultimately decrease their overall learning.

1.2 Student Created Cheat-Sheets

A student created cheat-sheet is simply a sheet of notes produced while preparing for an examination. The size of the sheet can be specified and a common size is double-sided A4. While students are generally free to add whatever information they believe is relevant, the production of the sheet may be constrained; for example a teacher may specify that the sheet must be hand-written. Forcing students to hand-write their cheat-sheet is a mechanism to ensure that students make some effort to produce the sheet, rather than simply printing course notes or photocopying another student's sheet.

The effect of student created cheat-sheets has been less well explored than open-book examinations, but work in this area is more recent. Dickson and Miller (2005) explored students' use of cheat-sheets in psychology examinations, finding evidence that suggested cheat-sheets did not improve performance and did not reduce student anxiety. Dickson and Miller later revisited student created cheat-sheets using a different experimental approach and focused on the suggestion that preparing such sheets may encourage learning. They allowed students to prepare cheat-sheets, and then at the examination Dickson and Miller removed the cheat-sheets from students and asked them to take the examination. After this, they returned the cheat-sheets and allowed the students to take the examination again. Results showed that students performed better when they had access to their cheat-sheets and from this they concluded that cheat-sheets did not encourage greater learning, but did assist students during an examination (Dickson & Miller, 2008). Dickson and Miller failed to take into account that cheat-sheets are intended to relieve students of the burden of memorisation, yet memorisation seems to be what their experiment was measuring.

Almost in complete opposition to Dickson and Miller is the work of Erbe (2007), who suggests that student created cheat-sheets can reduce examination anxiety while increasing learning, particularly in courses that assess on the first three levels of Bloom's taxonomy (Bloom, 1956). Erbe emphasises that examinations do more than assess learning; the way examinations are structured and implemented can cause student learning, a thought also shared by Yu, Tsiknis and Allen (2010). Erbe suggests that open-book examinations can cause students to be lulled "into too much of a sense of security and, if they had not prepared adequately, the book was not very useful anyway" (p. 97). Erbe quotes Boniface (1985) in relation to this. Erbe noticed variety in the content and composition of cheat-sheets constructed by her students and now awards a prize for the best cheat-sheet; doing this also helps to reduce the tension around the examination. Erbe states that while students spend a lot of time preparing their cheat-sheets, they do not actually refer to them often in the examination: "Preparing the cheat sheets proved to be sufficient for learning what was on the test. This was the major difference between handing out information composed by me and having the students find their own. Students tailored the information to their own needs and wrote down information they still needed to learn. The act of writing and organizing the information for the cheat sheet

allowed most students to fill in the holes in their knowledge" (p. 97). A number of other educationalists share the same view (Janick, 1990; Weimer, 1989), however there does not appear to be objective, empirical evidence to support this view.

There can be diversity in the quality and composition of student created cheat-sheets. Visco et al. (2007) analysed the cheat-sheets that students created for a chemical engineering examination. They found great variety among students' cheat-sheets and suggested that the "goodness" of a cheat-sheet does not necessarily map to examination performance.

1.3 Examinations in Computing Education

Since 1988 when Bagert asked the question: *Should computer science examinations contain "programming" problems?* (Bagert Jr., 1988), instructors of programming have been considering how students should be assessed, particularly in examinations. The nature and content of examinations are currently a topical issue in Computing Education research (Lister et al., 2004; Lister et al., 2006; Sheard et al., 2008; Sheard et al., 2011; Whalley et al., 2006). Related to the make-up of examinations are the conditions under which examinations are conducted, such as the length of examinations and what resources students have access to during examinations.

In a review of introductory programming assessment, Daly and Waldron (2004) suggest allowing "students to bring in a handwritten A4 'cheat-sheet' which can contain whatever they want. The process of creating the 'cheat-sheet' may also be educational" (p. 211). Daly and Waldron do not mention how they believe cheat-sheets are educational. No studies have reported on the use of cheat-sheets in computing education examinations.

1.4 Research Questions

In real-world circumstances, programmers rely on resources for specific information, such as syntax specifications and examples of solutions to problems. While expert programmers possess a wealth of tacit solutions to problems (Soloway, 1986), they are not expected to memorise specific information, so it is unrealistic to expect students to do so for an examination.

Student created cheat-sheets may overcome the need for memorisation and bring about other benefits, but this idea needs to be explored and analysed. To achieve this, the following research questions are proposed.

- Do students who create and use a cheat-sheet perform better than students who do not?
- Does a student created cheat-sheet lift a student's performance compared to earlier course assessment?
- What features can be identified on student created cheat-sheets?
- Do these identifiable features relate to examination performance?

2 Methodology

In order to answer the above research questions, an analysis is to be performed on the cheat-sheets created by students and used in an examination.

2.1 Setting

The examination was conducted at the end of an introductory programming course run at the University of Southern Queensland. There were 89 students who sat the examination including a mix of on-campus (21%) and external (79%). Students sat the examination at numerous examination centres around the world.

Leading up to the examination, students were provided with a sample examination that mirrored the style of the final examination but included different questions. The sample examination included a reference to language specific functions, similar to what students had used during the course, but customised for the examination.

Students were informed about the topics covered by questions in the examination. Both the sample examination and final examination included a mix of code writing and short answer questions. All questions were new and could not be answered by simply copying from a cheat-sheet.

Students were informed about the conditions of the examination, including the ability for them to bring a student created cheat-sheet. The cheat-sheet requirements were specified as:

- hand-written;
- A4, double-sided; and
- containing any information they saw as relevant.

Students were told that the final examination would include a language reference, similar to the sample examination, so they need not include such information in their cheat-sheets.

Students were required to submit their cheat-sheet with their examination papers. The cheat-sheets were collected by examination invigilators, who had specific instructions to do so.

From an experimental perspective, there was no mechanism for ensuring that there would be groups with and without cheat-sheets and there was no attempt to control the membership of these groups when they emerged.

2.2 Method of Analysis

The analysis of cheat-sheets was conducted by:

1. identifying each sheet with a code number,
2. separating them from their accompanying examination answers, and then
3. identifying features in each (see Coding Scheme section below), which were recorded against the code numbers.

All feature analysis was conducted before comparing the use of cheat-sheets, and the contained features, against student performance.

2.2.1 Coding Scheme

Before attempting to identify features in all cheat-sheets, a subset of the cheat-sheets was examined and a number of common features were identified. No pre-existing schema was used. These features are described in Table 1. There were two categories of features: those that related to layout (how information was organised on cheat-sheets) and content (what information was found on cheat-sheets).

The features were then checked in the entire collection of cheat-sheets. Each of the features was identified in a simple binary fashion, being either present or absent.

3 Results

From the 89 students who sat the examination 72 cheat-sheets were collected, which indicates that 81% of students chose to create a cheat-sheet and 19% of students either chose not to create a cheat-sheet or missed the fact that they could.

3.1 Relative Performance

Relative performance of students with and without cheat sheets was measured. There was no control over how students fell into these groups, however the range in both groups' marks was roughly equal, as indicated by the minima and maxima in Table 2.

<i>Layout Features</i>	Dense	A cheat-sheet was <i>dense</i> when both sides of the paper were covered, leaving little vacant space. This is a measure of the amount of information on the paper and possibly the effort invested in creating the sheet.
	Organised	A cheat-sheet was <i>organised</i> when the space on the paper was compartmentalised to make fuller use of space, usually with boundaries and identifiers for compartments.
	Order matches course content	If the ordering of the content on the student's cheat-sheet followed the ordering of content presented in the course, this feature was seen as present.
<i>Content Features</i>	Code examples	Relevant to a programming examination, the presence of program code examples in the cheat-sheet was measured.
	Abstract representations	Marked as present when concepts were represented in a general way, using text or diagrams rather than program code for specific examples.
	Sample answers	Where students included answers to sample examination questions, this feature was considered present.
	Language reference duplication	This feature was marked as present when students included information that duplicated information provided in the language reference in the paper.

Table 1: Features measured in cheat-sheets

	Min	Max	Mean	StdDev
<i>With cheat-sheet</i>	3	47	32.5	11.9
<i>Overall</i>	0	47	29.0	13.2
<i>Without cheat-sheet</i>	0	47	23.9	15.7

Table 2: Performance with and without cheat-sheets

In a t-test for equality of means, with roughly equal variance, there was a significant difference between the two groups ($t=2.5$, $p=0.016$). This indicates that students who produced and used a cheat-sheet performed significantly better than those who did not.

The overall average performance in the examination was 29.0 out of 50 possible marks. Students with cheat-sheets performed, on average, higher than this mean and students without cheat-sheets performed worse, as shown in Table 2.

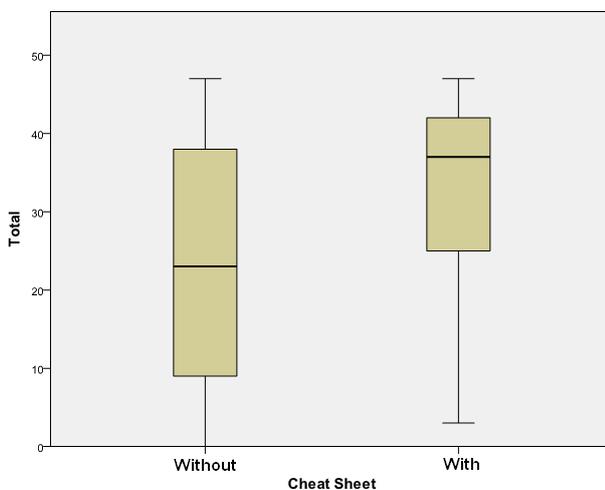


Figure 1: Comparison of performance by students with and without cheat-sheets

The distribution of marks by each group (with and without cheat-sheets) is represented in Figure 1. The box-plot on the left relates to students without cheat-sheets. This group had a lower mean at 23.9. The box-plot on the right shows the performance of students who created and used a cheat-sheet. This group had a higher mean at 32.5 and generally performed above a “passing” mark of 25, with a narrower standard deviation (see Table 2).

3.2 Improvement

Improvement was measured by comparing each student’s performance in prior assessment (relative to the mean prior assessments) and their performance in the examination (relative to the mean examination mark).

On average, students who used a cheat-sheet improved their performance, with the opposite effect demonstrated by students without cheat-sheets, as shown in Table 3. In other words, preparing and using a cheat-sheet helped students to improve their level of performance between prior assessment and the examination.

	Examination Marks Improvement	Overall Assessment Improvement
<i>With cheat-sheet</i>	+5.3	+2.6%
<i>Without cheat-sheet</i>	-2.2	-1.1%

Table 3: Improvement in performance by students with and without cheat-sheets

3.3 Identified Features

In the 72 cheat-sheets collected, features were identified. A total of seven common features were identified in students’ cheat-sheets. The list of features is given in the Coding Scheme section above, with a description of how each feature was identified. The occurrence of each feature is given in Table 4 together with the measured impact of each feature.

		Occurrences	Difference to mean
<i>Layout Features</i>	Dense	43 (60%)	+1%
	Organised	49 (68%)	-2%
	Order matches course content	14 (19%)	+13%
<i>Content Features</i>	Code examples	52 (72%)	-7%
	Abstract representations	34 (47%)	+21%
	Sample answers	15 (21%)	-30%
	Language reference duplication	7 (10%)	+3%

Table 4: Occurrence and impact of identified features

The analysis of features shows that the presence of some features relates to higher or lower examination performance, on average, by the creators of the cheat-sheets that contain them. Other features do not seem to relate to a difference in performance. The impact of each feature was calculated by comparing the average of the group of students whose cheat-sheets exhibited that feature with the overall mean and finding the difference.

In layout features, the density and organisation of information in a cheat-sheet did not seem to relate to any major difference in performance. Students who ordered the content of their cheat-sheets to match the ordering of course content performed, on average, 13% better than the mean.

The analysis of content features showed that students included code examples as well as abstract representations of content presented in the course. While some students included both (22%), most students included only one or the other. Students who included code examples in their cheat-sheets tended to perform slightly lower than average. Students who included abstract representations of concepts tended to perform 21% higher than the average.

Ignoring advice to the contrary, some students included information already provided in a language reference. Despite the fact that this content took up space that could have been dedicated to other content, such students did not seem to be negatively impacted by doing so.

4 Conclusions

The findings of this study indicate that the preparation and use of student created cheat-sheets does have an impact on student performance. This contradicts the findings of Dickson and Miller (2005) and provides evidence that supports the suggestions of Erbe (2007) and Daly and Waldron (2004) that the process of creating a cheat-sheet can improve student outcomes.

This study found variety in features in student's cheat-sheets, which echoes the findings of Visco et al. (2007). Ordering cheat-sheet content to match course content relates to higher examination performance. This may indicate that students who create cheat-sheets in such an ordered fashion are undertaking a more thorough, start-to-finish approach when creating their sheets, and perhaps learning more from this experience. It may also be the case that when content was ordered in a way that was familiar to the student's experience, less effort was required to find information on the cheat-sheet during the examination, which relates to the concern of Boniface (1985) who suggested that time used referring to information in examinations degrades performance.

Students who included abstract representations of content in their cheat-sheets were more successful. This may be due to abstract representations being more adaptable to new problems than specific examples. In order to include and use abstract representations students would need to have reached the higher SOLO relational or extended abstract levels (Biggs & Collis, 1982), while students who relied on coding examples may be working at the lower multistructural level. Answering at a higher SOLO level has been related to understanding in introductory programming (Lister et al., 2006).

The inclusion of sample examination answers in cheat-sheets was related to poorer performance. It seems likely that students who did this hoped that questions in the final examination would be the same as questions in the sample examination, which was not the case. This seems to reveal a poorer understanding of concepts by students. The sample examination was intended to be a test of student learning after revision and not the only instrument for revision itself.

Where students included material that duplicated what was available in the language reference included in the examination paper, there was no major impact on performance, even though the space used by this content could have been used for other, more necessary content. This perhaps indicates that revising such references may have aided student understanding, and having their own descriptions of such references may have made them easier to look up and use.

4.1 Recommendations

From the results of this study, it is recommended that examiners consider allowing students to use cheat-sheets

in examinations as this may increase learning, reduce anxiety and, as shown in the results of this study, lead to improved performance.

When creating their cheat-sheets, students should be advised to:

- conduct a thorough review and order the content of their sheet to match the ordering of course content;
- attempt to record generalised, abstract representations of concepts, rather than specific examples, so that ideas can be adapted during examinations; and
- avoid hoping that answers to sample examinations will match final examination questions.

4.2 Future Work

The findings of this study would benefit from wider analysis in other introductory programming courses with more control over experimental group membership. It would be interesting to see if the findings of this study continued longitudinally. For more generally applicable results, experimentation in other knowledge domains would be needed.

5 References

- Bacon, F. (1969): Open book Examinations. *Education and Training*, **11**(9):363.
- Bagert Jr., D. J. (1988): Should computer science examinations contain "programming" problems? *Proceedings of the nineteenth SIGCSE technical symposium on Computer science education (SIGCSE'88)*, Atlanta, Georgia, USA, February 25-26, 1988. 288 - 292.
- Biggs, J. B., & Collis, K. F. (1982): *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. New York, Academic Press.
- Bloom, B. S. (1956): *Taxonomy of Educational Objectives*, Edwards Bros., Ann Arbor, Michigan.
- Boniface, D. (1985): Candidates' use of notes and textbooks during an open-book examination. *Educational Research*, **27**(3):201 - 209.
- Daly, C., & Waldron, J. (2004): Assessing the Assessment of Programming Ability. *Proceedings of the 35th SIGCSE technical symposium on Computer science education (SIGCSE '04)*, Norfolk, Virginia, USA, March 3 - 7, 2004 210 - 213.
- Dickson, K. L., & Miller, M. D. (2005): Authorized Crib Cards Do Not Improve Exam Performance. *Teaching of Psychology*, **32**(4):230 - 233.
- Dickson, K. L., & Miller, M. D. (2008): Do Students Learn Course Material During Crib Sheet Construction? *Teaching of Psychology*, **35**(2):117 - 120.
- Erbe, B. (2007): Reducing Text Anxiety while Increasing Learning - The Cheat Sheet. *College Teaching*, **55**(3):96 - 98.
- Feldhusen, J. F. (1961): An Evaluation of College Students' Reactions to Open Book Examinations. *Educational and Psychological Measurement*, **21**(3):637 - 646.

- Janick, J. (1990): Crib Sheets. *Teaching Professor*, **4**(6):2.
- Kalish, R. A. (1958): An Experimental Evaluation of the Open Book Examination. *Journal of Educational Psychology*, **49**(4):200 - 204.
- Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., et al. (2004): A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, **36**(4):119 - 150.
- 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. *ACM SIGCSE Bulletin*, **38**(3):118 - 122.
- Schumacher, C. F., Buztin, D. W., Finberg, L., & Burg, F. D. (1978): The Effect of Open- vs. Closed-Book Testing on Performance on a Multiple-Choice Examination in Pediatrics. *Pediatrics*, **61**(2):256 - 261.
- Sheard, J., Carbone, A., Lister, R., & Simon, B. (2008): Going SOLO to assess novice programmers. *SIGCSE Bulletin*, **40**(3):209 - 213.
- Sheard, J., Simon, Carbone, A., Chinn, D., Laakso, M.-J., Clear, T., et al. (2011): Exploring Programming Assessment Instruments: a Classification Scheme for Examination Questions. *Proc. Proceedings of the International Conference on Educational Research (ICER) 2011*.
- Soloway, E. (1986): Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, **29**(9):850 - 858.
- Theophilides, C., & Koutselini, M. (2000): Study Behavior in the Closed-Book and the Open-Book Examination: A Comparative Analysis. *Educational Research and Evaluation*, **6**(4):379 - 393.
- Tussing, L. (1951): A Consideration of the Open Book Examination. *Educational and Psychological Measurement*, **11**(4):597 - 602.
- Visco, D., Swaminathan, S., Zagumny, L., & Anthony, H. (2007): Interpreting Student-Constructed Study Guides. *Proceedings of the 114th Annual ASEE Conference & Exposition*, Honolulu, Hawaii, USA, June 24 - 27, 2007. 1 - 9.
- Weimer, M. (1989): Exams: Alternate ideas and approaches. *Teaching Professor*, **3**(8):3 - 4.
- Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robins, P., Kumar, P. K. A., et al. (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, January 2006. 243 - 252.
- Yu, B., Tsiknis, G., & Allen, M. (2010): Turning Exams Into A Learning Experience. *Proc. Proceedings of the 41st ACM technical symposium on Computer science education (SIGCSE2010)*.

Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions

Malcolm Corney and Donna Teague

Faculty of Science and Technology
Queensland University of Technology,
Brisbane, QLD, Australia

{m.corney,d.teague}@qut.edu.au

Alireza Ahadi and Raymond Lister

Faculty of Engineering and Information Technology,
University of Technology, Sydney,
Sydney, NSW, Australia

Raymond.Lister@uts.edu.au

Abstract

Recent research on novice programmers has suggested that they pass through neo-Piagetian stages: sensorimotor, preoperational, and concrete operational stages, before eventually reaching programming competence at the formal operational stage. This paper presents empirical results in support of this neo-Piagetian perspective. The major novel contributions of this paper are empirical results for some exam questions aimed at testing novices for the concrete operational abilities to reason with quantities that are conserved, processes that are reversible, and properties that hold under transitive inference. While the questions we used had been proposed earlier by Lister, he did not present any data for how students performed on these questions. Our empirical results demonstrate that many students struggle to answer these problems, despite the apparent simplicity of these problems. We then compare student performance on these questions with their performance on six explain in plain English questions.

Keywords: Novice programmer, CS1, neo-Piagetian.

1 Introduction

It is well documented within the research literature that many CS1 students around the world struggle to learn to program. For example, McCracken's (2001) multi-national ITiCSE working group collected data from over 200 CS1 students. The students were required to write code to evaluate arithmetic expressions. The average student score was only 21%. Most tellingly, many of the students did not write any code, as they spent their allotted 90 minutes trying to come up with a design for the program. Inspired by the McCracken working group, the ITiCSE 2004 "Leeds" Working Group (Lister *et al.*, 2004) tested the reading and tracing skills of over 500 end-of-CS1 students, from twelve universities in seven countries. The average score for the students was 60%, with a quarter of the students performing at a level consistent with choosing options at random.

The literature on the novice programmer also abounds with reports on puzzling behaviours exhibited by novice programmers. For example, Thomas, Ratcliffe, and

Thomasson (2004) wrote about their frustrations at getting novices to use diagrams:

... when they might appropriately use [diagrams] themselves, weaker students fail to do so. They are often impatient when the instructor resorts to drawing a diagram, then amazed that the approach works. ... [also] providing [students] with what we considered to be helpful diagrams did not significantly appear to improve their understanding. ... This was completely unexpected. We thought that we were 'practically doing the question for them'...

Perkins *et al.* (1986) described the behaviours of students who are "movers" or "stoppers":

Many students disengage from the task whenever trouble occurs, neglect to track closely what their programs do by reading back the code as they write it, try to repair buggy programs by haphazardly tinkering with the code, or have difficulty breaking problems down into parts suitable for separate chunks of code.

Ginat (2007) described a similar approach to that of the tinkerer, but in more advanced students:

A hasty design may be based on some simplistic application of a familiar design pattern ... based ... on some premature association that seems relevant. Errors are not always discovered, as the test cases on which the program is tested are very limited. The devised program is batched, and "seems correct". Then, an outside source (e.g., a teacher) points out a falsifying input. A patch is offered. Sometimes the patch is sufficient for yielding correctness, but more often than not, the patch is insufficient. An additional patch is offered; and the cycle of batch-&-patch continues.

1.1 Overview

Results such as those cited above recently led Lister (2011) to describe a neo-Piagetian perspective of the novice programmer. In the next section of this paper, we review his neo-Piagetian perspective. In section 3 we report upon empirical results for some neo-Piagetian inspired exam questions suggested by Lister. Section 4 then examines the relationship of those questions to code explanation questions.

Copyright © 2012, Australian Computer Society, Inc. This paper appeared at the 14th Australasian Computing Education Conference (ACE 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 123. M. de Raadt and A. Carbone, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

2 Background: Neo-Piagetian Development

Lister (2011) proposed four stages of cognitive development of the novice programmer, based on neo-Piagetian theory. In the following subsections, we outline those neo-Piagetian stages. For a more detailed description, the reader should see Lister (2011).

2.1 Sensorimotor Stage

The first neo-Piagetian stage is the sensorimotor stage. Based upon the empirical results from Philpott, Robbins and Whalley (2007), Lister proposed that novices who trace code with less than 50% accuracy are at the sensorimotor stage. The afore-mentioned Leeds Working Group (Lister *et al.*, 2004) demonstrated that there certainly exist students who cannot trace code with 50% reliability at the end of their first semester of learning to program.

Without the ability to reliably produce consistent results via tracing, novices at the sensorimotor stage see code as somewhat magical. That is, they do not experience an executing program as a deterministic machine.

2.2 Preoperational Stage

At the next stage of development, the preoperational stage, novice programmers can reliably trace code, but they do not routinely abstract from the code to see a meaningful computation performed by that code. Again, the Leeds Working Group (Lister *et al.*, 2004) described students who were able to trace code reliably, but...

“... While working out their answer, none of these students volunteered any realization of the intent of the code ...” (p. 138).

Novice programmers at this stage are the novices that Thomas *et al.* (2004) wrote about:

“... providing [students] with what we considered to be helpful diagrams did not significantly appear to improve their understanding.”

For the preoperational novice, the lines in a piece of code are only weakly related. This stage in the development of the novice programmer is like the stage that Piaget identified in a child's understanding of machines, such as bicycles, where the various parts are known to be necessary, but how the parts work together is not understood (Piaget, 1930, pp. 205–210). In an interview extract given in Traynor, Bergin, and Gibson (2006) a student described his approach to answering coding questions in an exam:

“... you usually get the marks by 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”.

That student quoted by Traynor *et al.* was perhaps being cynical, but in the context of this paper, that student is describing all that a preoperational novice can do when they are required to write code – put down the elements that they recognise must be there, but not be able to fit

those elements together in a way that produces correct code.

Without being able to see how the lines in a piece of code relate, a novice at the preoperational stage is likely to struggle with describing the purpose of a piece of code (“explaining”).

2.3 Concrete Operational Stage

Unlike students at the preoperational stage, students at the concrete operational stage can reason about abstractions of their code. They can, for instance, relate code to diagrams. They can also see how the individual lines in a piece of code work together to perform some overall computation. However, a defining characteristic of concrete thinking is that the abstract thinking is restricted to familiar situations (hence “concrete”).

The three archetypal manifestations of concrete thinking are the abilities to reason (1) about processes that are reversible, (2) with quantities that are conserved and (3) properties that hold under transitive inference. In the next three subsections, we review three exam questions that Lister (2011) identified as requiring these three types of reasoning.

2.3.1 Reversing

Figure 1 contains a question that Lister (2011) nominated as requiring the novice programmer to reason about reversing.

The purpose of the following code is to move all elements of the array *x* one place to the **right**, with the **rightmost** element being moved to the **leftmost** position:

```
int temp = x[x.length-1];
for (int i=x.length-2; i>=0; --i)
    x[i+1] = x[i];
x[0] = temp;
```

Write code that undoes the effect of the above code. That is, write code to move all elements of the array *x* one place to the **left**, with the **leftmost** element being moved to the **rightmost** position.

Figure 1: A question that requires the concrete operational ability to reason about reversing (from Lister, 2011).

2.3.2 Conservation

Lister (2011) identified one type of conservation in programming, which is the preservation of a specification across variation in the implementation. Figure 2 contains a question he nominated as requiring the novice programmer to reason about conservation. In that question, either of the options in each box could be right, depending upon what choices the novice has made in the other boxes. Thus, the novice needs to be able to see how the lines of code are related.

Below is incomplete code for a method which returns the smallest value in the array `x`. The code scans across the array, using the variable `minsofar` to remember the smallest value seen thus far. There are two ways to implement remembering the smallest value seen thus far: (1) remember the actual value, or (2) remember the value's position in the array. Each box below contains two lines of code, one for implementation (1), the other for implementation (2). First, make a choice about which implementation you will use (it doesn't matter which). Then, for each box, draw a circle around the appropriate line of code so that the method will correctly return the smallest value in the array.

```
public int min(int x[] ){
    int minsofar = (a) 0  
(b) x[0] ;
    for ( int i=1 ; i<x.length ; ++i )
    {
        if ( x[i] < (c) minsofar  
(d) x[minsofar] )
            minsofar = (e) i  
(f) x[i] ;
    }
    return (g) minsofar  
(h) x[minsofar] ;
}
```

Figure 2: A question that requires the concrete operational ability to reason about conservation of specification under variation of implementation (from Lister, 2011).

2.3.3 Transitive Inference

Transitive inference is the type of reasoning where, in general terms, if a certain relationship holds between object A and object B, and if the same relationship holds between object B and object C, then the same relationship also holds between object A and object C. For example, Piaget would sometimes ask a child a question like, "If Adam is taller than Bob, and Bob is taller than Charlie, who is the tallest?"

Figure 3 contains the "explain in plain English" problem used in many BRACElet studies (Whalley *et al.*, 2006; Lister *et al.*, 2006). Lister (2011) nominated this question as requiring the novice programmer to perform transitive inference, since the novice must realise that if all consecutive array element pairs are ordered, then the entire array is ordered.

2.4 Formal Operational Stage

The formal operational stage is the most advanced and most abstract stage of cognitive development. It can be defined succinctly thus: formal operational reasoning is what competent programmers do, and what we'd like our students to do.

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 3: A question from several BRACElet studies, which requires the concrete operational ability of transitive inference (from Lister, 2011).

This paper focuses on the types of reasoning that precede formal operational reasoning, so in this paper it is only necessary to further sharpen the reader's understanding of concrete operational reasoning by describing how people who reason at the formal operational level differ from people reasoning at the concrete operational reasoning:

- They can reason about unfamiliar situations.
- They tend to begin with the abstract and move to the concrete.
- They reason with abstractions routinely, logically, consistently and systematically.
- They have a reflective capacity — an ability to think about their own thinking.
- They can perform hypothetico-deductive reasoning. In the context of programming, hypothetico-deductive reasoning is nicely illustrated by an extract from Edwards (2004), in a paper where he argued that novice programmers needed...

"... practice in hypothesizing about the behavior of their programs and then experimentally verifying (or invalidating) their hypotheses. ... These activities are at the heart of software testing." (p. 27)

- They reliably manifest problems solving skills on unfamiliar problems. McCracken *et al.* (2001) defined problem solving as a five step process: (1) abstract the problem from its description; (2) generate subproblems; (3) transform subproblems into subsolutions; (4) recompose; and (5) evaluate and iterate.

2.5 Note: Development vs. Pedagogy

The above neo-Piagetian stages do not imply pedagogy. For example, these stages do not imply that a novice should first be taught to trace code, before the novice is allowed to write any code. The above neo-Piagetian stages are descriptions of the order in which a novice's competence in certain skills will be manifested, irrespective of how that novice is taught.

3 Results for Reversing and Conserving

We placed into our end of semester exam the questions in Figures 1 and 2 which Lister (2011) had proposed as requiring concrete operational reasoning. This section discusses the results for those two questions.

3.1 Screening

Before performing the analysis below, we screened students, using two tracing questions from that same end of semester exam. The purpose in the screening was to eliminate from further study any students who were at the sensorimotor stage. Students who answered either of the two tracing questions incorrectly were eliminated from further study.

One of the two screening questions required students to determine the final values in five variables after a series of ten assignment statements. The first five assignment statements initialised each variable. The remaining five statements assigned values between these variables, and were designed to detect students who had any of the well known misconceptions about variables and assignment statements (du Boulay, 1988).

The other screening question required students to reason about two nested `if` statements. The conditions in the `if` statements involved comparisons among three integer variables, `a`, `b` and `c`. Each `then` and `else` part of the `if` statements results in the output of one of those variables. The question was framed as a multiple choice question, where students had to reason backward, from output to input. Specifically, students were asked “Which of the following values for the variables will cause the value in variable `b` to be printed?”

After this screening, 93 students remained in the sample for further analysis. These 93 students were considered to be reasoning at a level no lower than preoperational.

3.2 Reversing

When writing the solution to the problem in Figure 1, the student must recognise that the assignment `x[i+1] = x[i]` in the loop body needs to be replaced with either `x[i] = x[i+1]` or `x[i-1] = x[i]`. We feel that such a change is the simplest of all the changes required, and is even a change within the grasp of any exam-savvy student reasoning at the preoperational level. Rather than indicating a low level of neo-Piagetian reasoning, an error on that line of code might simply be due to a student misunderstanding the question, perhaps because of poor English language reading skills. Therefore, we eliminated from the analysis of this question any student who did not make a correct change to that assignment statement in the loop body, which left us with 70 students in our sample. All of these students provided a four-line solution that resembled the code provided in the question.

Of the 70 students, only 45 (64%) provided a correct first line, in which they saved the leftmost element of the array to the temporary variable, and 38 students (54%) provided a correct final line, in which they assigned the temporary value to the rightmost position in the array. Only 37 students (53%) provided both a correct first line and a correct last line. We classify the 33 students (47%) who did not provide correct versions of both lines as

clearly exhibiting preoperational reasoning. (Recall that these 33 students did provide a correct assignment in the loop body, and thus showed some understanding of the problem.)

Of the 37 students who provided a correct first, third and fourth line, 15 (41%) provided a correct version of the second line, the `for` loop. We classify those 15 students as clearly exhibiting concrete operational reasoning. We consider the remaining 22 of these 37 students to be exhibiting some degree of concrete operational reasoning. Among these 22 students, the most common errors were off-by-one errors. Often, the values specified in line 2 through which the control loop variable `i` would iterate were appropriate, in isolation, and so was the assignment statement on line 3, in the body of the loop. However, those two lines, in combination, were often not compatible. Perhaps with a little more careful checking, at least some of those students might have provided a correct solution.

In summary, for this question we see evidence for preoperational and concrete operational reasoning among our sample of students, who had passed a screen for sensorimotor reasoning.

3.3 Conservation

Table 1 shows student performance in the final exam on the concrete operational “choose from each box” task shown in Figure 2. The 40% of students who provided a correct solution (i.e. either ADEH or BCFG) are clearly exhibiting concrete operational reasoning. The 32% of students who provided ACFG show some signs of concrete operational reasoning, by virtue of choosing CFG. The remaining 28% of students are clearly exhibiting preoperational reasoning.

A possible threat to the validity of this question is the lengthy English instructions prior to the code. A student who reads English as a second language may be disadvantaged.

ADEH (correct)	8 %
BCFG (correct)	32 %
ACFG (close)	32 %
Others (all wrong)	28 %

Table 1: Student performance in the final exam on the concrete operational “choose from each box” task shown in Figure 2. (n=93)

3.4 Comparing Reversing and Conservation

Table 2 shows the relationship between student performance on these two questions, as a contingency table. Given that few students answered the code reversal problem with complete accuracy, we elected to just use the data on that question for how many students handled the end element correctly (i.e. lines 1 and 4). A χ^2 test yielded $p=0.08$, which is higher than the traditional $p=0.05$ threshold for statistical significance.

With the given data, it is unclear whether the absence of a statistical relationship between the two questions is due to competence differences (i.e. the two questions

require different sorts of reasoning skills) or performance differences (i.e. the framing of the two questions test skills other than the ability to reason about code; see Chomsky, 1965). One obvious potential performance difference is the greater demands placed upon a student’s English language reading ability by the “select from the boxes” task. A more detailed study of this issue is warranted. An essential element of such a study is the use of two or more questions of each type, to assess the consistency of student performance within each type of question, before assessing the significance of consistency of student performance between these two types of questions.

Select code from boxes (see Figure 2)	Write the reverse of a given shift; correct treatment of end element (i.e. correct line 1 and line 4, analogous to Figure 1)		
	wrong	right	
wrong	36	20	56
right	17	20	37
	53	40	93

Table 2: The contingency table for student performance on the two concrete operational questions. ($\chi^2 = 3.06, p = 0.08$)

4 Code Explanation and Concrete Reasoning

This section explores the relationships between the two concrete operational questions studied in the previous section and explain in plain English questions.

Explain in plain English questions were used extensively in the BRACElet project (Whalley, *et al.*, 2006; Lister *et al.*, 2006). However there has been some controversy as to whether these questions are really testing the ability of students to read and understand code (i.e. competence) or the ability of the students to express themselves in English (i.e. performance; see Simon, *et al.*, 2009; Simon, 2009; Simon and Snowden, 2011). If we see in our data a direct relationship between how well our students answer the concrete operational questions in Figures 1 and 2 and how well they answer explain in plain English questions, then that would suggest that the explain in plain English question requires, at the minimum, concrete operational reasoning skills.

4.1 The Six Explanation Questions

Our end of semester exam contained six explain in plain English questions. These questions, labelled (a) to (f) are shown in Figures 4 and 5, along with handwritten answers from a student who did the exam:

- (a) This question was intended to be a benchmark of each student’s ability to express themselves in English. Since the code does not contain either loops or arrays, it is simpler than the remaining explanation questions.
- (b) This question was intended to test the student’s ability to explain code operating on arrays, using an `if` statement within a loop.

- (c) In addition to the skills required to explain the previous question, this question required the student to reason about the effect of a `return` statement within a loop. The purpose in including this question was that it requires the same skills as question (f), with the exception of transitive inference.
- (d) We regard this as our simplest question on loops and arrays.
- (e) The basic purpose of this code is relatively simple — to find the position of a target value in a list. However, there are two details required in a completely correct answer. The first detail concerns the behaviour of the code if the target value is not found. The second detail concerns the behaviour of the code when the target value occurs more than once in the list.
- (f) This question is similar to the question in Figure 3, that Lister asserted required transitive inference. However, our code in the exam was different from Figure 3 in two ways. The first difference is that the code we used in our exam was in Python. The second difference is our Python code does not set a flag, but instead (as in question (c)) breaks out of the loop with a `return` statement. A comparison of student performance on questions (c) and (f) is a test of Lister’s assertion that this question requires transitive inference.

4.2 Reversing and Explanation Questions

Row 1 of Table 3 shows the performance of the 93 students who passed our sensorimotor screening test, which was described earlier. For example, the column headed “(a)” shows that 89% of the whole sample answered the explanation question “(a)” correctly. The two hardest explanation questions (by far), were explanation questions (c) and (f), with only 36% and 31% of our whole sample answering those questions correctly. (Note that students’ answers to question (f) were marked as correct even if they failed to mention the indexing error that may be generated.) The remaining rows of Table 3 show the percentage of students who answered correctly each explanation question, given their performance on aspects of the concrete operational “shift left” question in Figure 1.

Row 2 of Table 3 shows the performance of the 33 students who (in addition to passing the sensorimotor screening test) provided a correct assignment statement within the body of the loop, but failed to provide a correct handling of the end element of the array (i.e. code like lines 1 and 4 in Figure 1). Only a quarter of these students could answer explanation questions (c) and (f) correctly. Less than half (45%) of these students answered explanation question (e) correctly.

Earlier in the paper, we surmised that writing the correct assignment in the `for` loop lay within the grasp of exam-savvy students reasoning at the preoperational level. The statistical data for Row 2 is consistent with that claim. Over half the students in Row 2 are able to correctly answer the two explanation questions (b) and (d). To do so, we believe a student need not understand

QUESTION 18**[6 marks]**

This question contains a series of functions. For each function, describe the purpose of the function in one sentence that you should write in the corresponding box below that function. Do **NOT** give a line-by-line description of what the code does. Instead, tell us the purpose of the code:

(a) (1 mark)

```
def function_a(x, y):
    if x > y:
        return x
    else:
        return y
```

Returns the bigger value of x or y, or y if they are equal.

(b) (1 mark)

```
# x and y are lists of equal length
def function_b(x, y):
    c = 0
    for index in range(len(x)):
        if x[index] > y[index]:
            c = c + 1
    return c
```

Compares the list values at the same positions and returns the amount of numbers, where the value in list x is bigger.

(c) (1 mark)

```
# x and y are lists of equal length
def function_c(x, y):
    for index in range(len(x)):
        if x[index] > y[index]:
            return True
    return False
```

Goes through the list and checks if there is once a bigger value in x ~~at any~~ then in y (at the same index).

Question 18 continued overleaf...

Figure 4: The first three of the six explanation questions used in the exam paper.

Question 18 continued

(d)

(1 mark)

```
# numbers is a list
def function_d(numbers):

    t = 0
    for index in range(len(numbers)):
        t = t + numbers[index]

    return t
```

Sums up all the values in the list and returns them. ~~Return~~
(sum(numbers))

(e)

(1 mark)

```
# numbers is a list, target is a single value
def function_e(numbers, target):

    p = -1
    for index in range(len(numbers)):
        if numbers[index] == target:
            p = index

    return p
```

Returns the latest index position of target or -1 if target is not in numbers.

(f)

(1 mark)

```
# numbers is a list
def function_e(numbers):

    for index in range(len(numbers)):
        if numbers[index] > numbers[index + 1]:
            return False

    return True
```

Checks if the list is sorted from low ~~low~~ to high values.

Figure 5: The final three of the six explanation questions used in the exam paper.

Row	Description	n	Explain in Plain English Questions (a) to (f)					
			(a)	(b)	(c)	(d)	(e)	(f)
1	Whole sample	93	89%	76%	36%	82%	60%	31%
2	Correct assignment in for loop	33	82%	58%	24%	67%	45%	24%
3			p < 0.04*	p < 0.01*	p = 0.8	p = 0.01*	p = 0.01*	p = 0.3
4	End element handled correctly	22	100%	95%	27%	95%	68%	23%
5			p = 0.22	p = 0.78	p < 0.01*	p = 0.78	p = 0.20	p < 0.03*
6	Code entirely correct	15	93%	93%	73%	93%	87%	60%

Table 3: The percentages of students who answered each explain in plain English question correctly, given their performance on aspects of the concrete operational “shift left” question in Figure 1. The shaded cells indicate statistically significant differences in the percentages shown in the cells above and below the shaded cell.

Explain in Plain English Questions (a) to (f)												Σ
(a)		(b)		(c)		(d)		(e)		(f)		
w	right	w	right	w	right	w	right	w	right	w	right	
6	27 (82%)	14	19 (58%)	25	8 (24%)	11	22 (67%)	18	15 (45%)	29	4 (12%)	33
0	22(100%)	1	21 (95%)	16	6 (27%)	1	21 (95%)	7	15 (68%)	17	5 (23%)	22
6	49 (89%)	15	40 (73%)	41	14 (25%)	12	43 (78%)	25	30 (55%)	46	9 (16%)	55
p < 0.04*		p < 0.01*		p = 0.8		p = 0.01*		p = 0.01*		p = 0.3		

Table 4: Complete contingency tables used to calculate the χ^2 test probabilities in row 3 of Table 3. The column heading “w” (wrong) indicates data for students who answered that particular explanation question incorrectly.

Explain in Plain English Questions (a) to (f)												Σ
(a)		(b)		(c)		(d)		(e)		(f)		
w	right	w	right	w	right	w	right	w	right	w	right	
0	22(100%)	1	21 (95%)	16	6 (27%)	1	21 (95%)	7	15 (68%)	17	5 (23%)	22
1	14 (93%)	1	14 (93%)	4	11 (73%)	1	14 (93%)	2	13 (87%)	6	9 (60%)	15
1	36 (97%)	2	35 (95%)	20	17 (46%)	2	35 (95%)	9	28 (76%)	23	14 (38%)	37
p = 0.22		p = 0.78		p < 0.01*		p = 0.78		p = 0.20		p < 0.03*		

Table 5: Complete contingency tables used to calculate the χ^2 test probabilities in row 5 of Table 3.

		Explain in plain English question (f)		Σ
		wrong	right	
Explain in plain English question (c)	wrong	3	1	4
	right	3	8	11
		6	9	15

Table 6: Contingency table comparing the performance on explanation questions (c) and (f) of the n=15 students who answered entirely correctly the concrete operational “shift left” question in Figure 1 (χ^2 test, p = 0.1).

	Description	n	Explain in Plain English Questions (a) to (f)					
			(a)	(b)	(c)	(d)	(e)	(f)
4	Handled end element	22	100%	95%	27%	95%	68%	23%
5	Select boxes correct	37	92%	92%	51%	86%	73%	41%
6	Entire shift correct	15	93%	93%	73%	93%	87%	60%

Table 7: The middle row shows the percentages of students who answered each explain in plain English question correctly, given correct performance on the “select from the boxes” question in Figure 2. The rows beginning “4” and “6” are the same rows as in Table 3, for comparison.

every aspect of that code. Like the student we quoted earlier from the paper by Traynor *et al.*, who knew how to get half marks on a code writing task, without understanding the code he put down, a student doesn't need to understand every aspect of the `for` statements in (b) or (d) to guess that the code will run across all elements of the list. After making such an assumption, a student can then answer the question by focussing solely upon the `if` within the loop, and its associated assignment statement.

4.2.1 Rows 2 & 4: Preoperational to Concrete

Row 4 of Table 3 shows the performance of the 22 students who succeeded at all the same tasks on the "shift" problem that the students in Row 2 were able to do, and who were also able to correctly handle the end element of the list (i.e. lines like 1 and 4 in Figure 1). However, these 22 students did not provide a suitable `for` statement (i.e. a line like line 2 in Figure 1). As discussed in section 3.2, these 22 students comprising row 4 exhibit some degree of concrete operational reasoning, whereas the students in row 2 exhibit preoperational reasoning.

A chi-square test was performed on each of the data forming rows 2 and 4 of the table. That is, for each explanation question, the raw data from which the two percentages in rows 2 and 4 were derived were used to perform a chi-square test. The resultant probability values for each column are shown in Row 3. (To assist others who may attempt to replicate our findings, the full contingency tables from which the probabilities were calculated are provided in Table 4.)

The shaded cells in Row 3 indicate the explanation questions for which the percentage in Row 2 is significantly different (i.e. $p < 0.05$) to the percentage in Row 4. All four of the easier explanation questions show a statistically significant improvement from Row 2 to Row 4. However, the two harder explanation questions (i.e. c and f) do not show a statistically significant improvement. Also, although the percentage of explanation question (e) rises between Rows 2 and 4 (from 45% to 68%), even with that increase almost one third of the students who could correctly handle the end element in the "shift" problem could not answer this explanation question. While explanation questions (b) and (d) are too difficult for quite a large percentage of the students in Row 2, these two questions were answered correctly by almost every student in Row 4.

To summarize this subsection: some degree of concrete operational reasoning tends to be both necessary and sufficient for answering correctly explanation questions (b), (d) and (e).

4.2.2 Rows 4 & 6: Growing Concrete Skills

Row 6 of Table 3 shows the performance of the 15 students who were able to provide a completely correct solution to the "shift left" problem. These 15 students comprising row 6 exhibit solid concrete operational reasoning, perhaps even formal operational reasoning.

A chi-square test was performed on each of the data forming rows 4 and 6 of the table. (As before, to assist

others who may attempt to replicate our findings, full contingency tables for calculating these probabilities are provided in Table 5.) The resultant probability values for each column are shown in Row 5 of Table 3. The shaded cells in Row 5 indicate a statistically significant improvement on the two harder explanation questions (i.e. c and f) from Row 4 to Row 6.

A substantial minority of Row 6 students cannot answer these explanation questions. From our reading of incorrect student responses to (c), we conclude that many of these students did not understand that executing a `return` statement within a loop will immediately terminate the loop. Such a weak grasp of the `return` statement is consistent with Lister *et al.* (2004), who reported that misconceptions about `return` statements were the only misconceptions observed in that study.

4.2.3 Transitive Inference and Explanation

Recall that the purpose of explanation question (c) was that it required the same reasoning skills as question (f), with the exception of transitive inference. In this subsection, we compare student performance on those two explanation questions.

Student performance on both (c) and (f) is so poor that it is difficult to make any comparisons. In Row 2 of Table 3, only 24% of the students answered each of (c) and (f) correctly. In Row 4 of Table 3, only 27% and 23% of the students answered (c) and (f) correctly. These two explanation questions are too hard for most students represented in those two rows.

Row 6 of Table 3 is the only row where any comparison of (c) and (f) is at all viable. Here, the respective percentages are 73% and 60%, but that is for a tiny sample of only 15 students. Table 6 is a contingency table for that data. The resultant probability value is $p = 0.1$, which is above the standard $p = 0.05$ threshold. However, $p = 0.1$ does mean that the chance that the difference in the percentages is a statistical fluke is only 1 in 10. Given that, and the small sample size, our data does not fundamentally contradict Lister's assertions about transitive inference, but it is at best only very weakly supportive. Further work is warranted.

4.3 Conservation and Explanation

Row 5 in Table 7 shows the percentages of students who answered each explanation question correctly, given correct performance on the "select from the boxes" question in Figure 2. The rows beginning "4" and "6" are the same rows as in Table 3, from the "shift" problem, for comparison.

Of particular interest in Table 7 is the data for the two harder explanation questions, (c) and (f). The percentages for "select from the boxes" on questions (c) and (f) lie between the percentages for the rows beginning "4" and "6", from the "shift" problem — which may indicate that the difficulty of this particular conservation problem is beyond some concrete operational students who can handle the end element in the "shift" problem, but lies within the grasp of most concrete operational students who can correctly solve the entire "shift" problem. That, in turn suggests (but does

not prove) that the “select from the boxes” task used in this paper requires concrete operational reasoning.

5 Conclusion

Our empirical results support the claims made by Lister (2011). We see students who manifest preoperational reasoning skills by their poor performance on a reversal task (“shift left”) and a conservation task (“fill in the boxes”). However, while our data does not fundamentally contradict Lister’s assertions about transitive inference, our limited data is at best only very weakly supportive. For transitive inference, a larger study will be required.

While there may be some controversy as to whether the nature of the problem that students face with explanation questions is competence-related or performance-related, there is less doubt about students who, when explicitly supplied with code that moves all elements of a list (or array) one place to the right, cannot alter that supplied code to move all the elements one place to the left. Thomas *et al.* (2004) described the diagrams they gave their students as “*practically doing the question for them*” — how much more so for the “shift left” problem we gave our students? Such a question clearly establishes that there are students in our class who, at the end of their first semester of programming, are at the preoperational level of reasoning about code. It would be very interesting to see if the same is the case at other universities — we suspect that it is the case.

Preoperational students are woefully under-prepared for the rigours of traditional programming assignments. On such assignments, preoperational students can only flail about, exhibiting the behaviours described by Perkins *et al.* (1986), Thomas *et al.* (2004), and Ginat (2007) — behaviours which have puzzled and exasperated many, many CS1 teachers around the world. A neo-Piagetian perspective on the novice programmer actually positions these behaviours as normal behaviours to be expected in the long and torturous cognitive development of the novice programmer.

References

- Chomsky, N. (1965). *Aspects of the Theory of Syntax*. Cambridge, MA: MIT Press.
- Du Boulay, B. (1988) *Some Difficulties in Learning to Program*. In Studying the Novice Programmer, Soloway, E. and Spohrer, J. C. (eds), Lawrence Erlbaum, 1988, pp. 283–299.
- Edwards, S. (2004) *Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action*. SIGCSE Bulletin 36, 1, 26–30.
- Ginat, D. (2007). *Hasty Design, Futile Patching and the Elaboration of Rigor*. SIGCSE Bull. 39, 3 (June), pp. 161–165.
- 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 Bull. 36, 4 (June), pp. 119–150. <http://doi.acm.org/10.1145/1041624.1041673>
- Lister, R., Simon, B., Thompson, E., Whalley, J., & Prasad, C. (2006). *Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy*. SIGCSE Bulletin 38(3): pp. 118–122.
- Lister, R. (2011) *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. Thirteenth Australasian Computing Education Conference (ACE 2011), Perth, Australia, January 2011. pp. 9–18.
- McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagen, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. (2001) *A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students*. SIGCSE Bull., 33(4). pp. 125–140.
- Perkins, D., Hancock, C., Hobbs, R., Martin, F., and Simmons, R. (1986) *Conditions of Learning in Novice Programmers*. In Soloway and Spohrer (eds), Studying the novice programmer. Lawrence Erlbaum. pp. 261–279.
- Philpott, A, Robbins, P., and Whalley, J. (2007): *Accessing The Steps on the Road to Relational Thinking*. 20th Annual Conference of the National Advisory Committee on Computing Qualifications, Nelson, New Zealand, p. 286.
- Piaget, J. (1930) *The Child's Conception of Physical Causality*. London, K. Paul, Trench, Trubner & Co.
- Simon, Lopez, M., Sutton, K. and Clear, T. (2009). *Surely We Must Learn to Read before We Learn to Write!*. In Proc. Eleventh Australasian Computing Education Conference (ACE 2009), Wellington, New Zealand. CRPIT, 95. Hamilton, M. and Clear, T., Eds., ACS. pp. 165–170.
- Simon (2009). *A Note on Code-Explaining Examination Questions*. Ninth International Conference on Computing Education Research – Koli Calling 2009, Koli, Finland, November 2009, pp. 21–30.
- Simon and Snowdon, S. (2011) *Explaining Program Code: Giving Students the Answer Helps – But Only Just*. Seventh International Computing Education Research Workshop (ICER), Providence, Rhode Island, pp. 93–99.
- Thomas, L., Ratcliffe, M., and Thomasson, N. (2004) *Scaffolding with Object Diagrams in First Year Programming Classes: Some Unexpected Results*. 35th SIGCSE technical symposium on Computer science education (SIGCSE '04). pp. 250–254.
- Traynor, D., Bergin, S., and Gibson, J. P. (2006) *Automated Assessment in CS1*. 8th Australian Conference on Computing Education (ACE), Hobart, Australia, ACM International Conference Proceeding Series, 165: pp. 223–228. <http://crpit.com/confpapers/CRPITV52Traynor.pdf>
- Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar A. P. K. & Prasad, C., (2006): *An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies*. Proceedings of the 8th Australasian Computing Education Conference, pp. 243–252.

Swapping as the “Hello World” of Relational Reasoning: Replications, Reflections and Extensions

Donna Teague and Malcolm Corney

Faculty of Science and Technology
Queensland University of Technology,
Brisbane, QLD, Australia

{d.teague,m.corney}@qut.edu.au

Alireza Ahadi and Raymond Lister

Faculty of Engineering and Information Technology,
University of Technology, Sydney,
Sydney, NSW, Australia

Raymond.Lister@uts.edu.au

Abstract

At the previous conference in this series, Corney, Lister and Teague presented research results showing relationships between code writing, code tracing and code explaining, from as early as week 3 of semester. We concluded that the problems some students face in learning to program start very early in the semester. In this paper we report on our replication of that experiment, at two institutions, where one is the same as the original institution. In some cases, we did not find the same relationship between explaining code and writing code, but we believe this was because our teachers discussed the code in lectures between the two tests. Apart from that exception, our replication results at both institutions are consistent with our original study.

Keywords: Novice programmer, tracing, explaining, writing.

1 Introduction

A number of recent research results have demonstrated a relationship between the ability of novice programmers to manually execute (“desk check” or “trace”) code, their ability to explain the purpose of a piece of code, and their ability to write similar code. Lopez et al. (2008) found that, when tracing and explaining were each used separately in a single regression model, neither tracing code nor explaining code were strong indicators of code writing ability. However, when combined in a multiple regression, tracing code and explaining code accounted for 46% of the variance in marks awarded to a code writing question in an exam. Venables, Tan and Lister (2009) performed a similar study, and also found a strong relationship between tracing, explaining and writing, as illustrated in Figure 1.

Lister, Fidge and Teague (2009) also studied the relationship between tracing, explaining and writing, but they used a non-parametric approach. As part of their results, they effectively screened students on their code tracing ability, which allowed them to isolate and study the relationship between code explaining and code

writing for a sample of students with a tracing performance >50%. For those students, Lister, Fidge and Teague found that students with ≤50% score on code explaining tasks performed statistically worse on a code writing task than students who scored >50% on the code explaining tasks (see Table 1). Similar results have since been reported for students at other educational institutions (Lister et al., 2010). From these studies, it seems plausible (but not proven) that a student is ill prepared to write code if that student also does not have reliable code tracing skills, or code explanation skills. If asked to design and write code, such a student may have little alternative but to engage in programming by “random mutation”, as the student may lack the analytic skills necessary to systematically debug their own code.

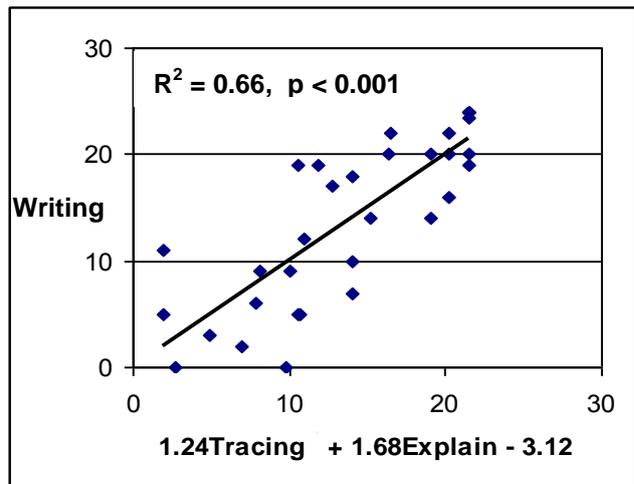


Figure 1: A multiple regression, from Venables, Tan and Lister (2009), with score on code writing as the dependent variable, and the combination of scores on tracing and explaining as the independent variables.

Number of good answers on four explanation questions	Success on a code writing question
> 50% (n = 98)	67%
≤ 50% (n = 24)	46%
χ^2 test	p = 0.05

Table 1: Empirical results from Lister, Fidge and Teague (2009), showing the relationship they found between code explaining and code writing. (This table is derived from Table 7 of their paper.)

Copyright © 2012, Australian Computer Society, Inc. This paper appeared at the 14th Australasian Computing Education Conference (ACE 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 123. M. de Raadt and A. Carbone, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

1.1 Corney, Lister and Teague (2011)

The above empirical studies all collected data from students at the end of their first semester of learning to program. In earlier work (Corney, Lister and Teague, 2011), we tested a class of CS1 students at three points in their development – at week 3, again at week 5, and at the end of semester. One of the questions in the week 3 test required students to answer the code explanation question shown in Figure 2.

The purpose of the following three lines of code is to swap the values in variables a and b:

```
c = a
a = b
b = c
```

The three lines of code below are the same as the lines above, but in a different order:

```
a = b
b = c
c = a
```

In one sentence that you should write in the box below, describe the purpose of those second set of three lines. **NOTE:** Tell us what the second set of three lines of code do all by themselves. Do NOT think of those second three lines as being executed after the first three lines of code.

Sample answer: “it swaps the values in b and c”

Figure 2: A question from the week 3 test of Corney, Lister and Teague (2011).

In the week 5 test, one of the questions required students to write code to swap the values of two variables (i.e. code similar to that shown in Figure 2). After eliminating from that sample those students who had performed poorly on some code tracing tasks, we found that students who had successfully explained the above swapping code in week 3 were much more likely to write correct code for swapping two variables in week 5 than the students who could not explain the code in week 3. In addition, the students who performed better on these questions in week 3 and week 5 performed better on a code writing task at the end of the semester. These results led us to conclude that the problems some students face in learning to program begin very early in semester.

1.2 Overview

In this paper, we present replications of our earlier work (Corney, Lister and Teague, 2011), performed at two institutions. One of the institutions is the Queensland University of Technology (QUT), which was the source of the data in the original study. The other institution is the University of Technology, Sydney (UTS). The QUT replication is very similar to the original study, while the UTS replication contains some variations from the original study.

2 Replication at QUT

In the replication at QUT, students were tested at week 3 and week 5 of semester, the same weeks as in the original study. As in the original study, these students were learning Python. In both weeks, we used the same test questions as in our original study. Also, we screened for and eliminated novices, using the same tracing questions as in our original study. After that screening, 51 students remained in our sample.

2.1 Writing a Swap

Table 2 summarises the results from this replication. The percentages shown in the brackets (and preceded by “cf.”) are the percentages from our original study. Our replication results do not support the results in our original study. The most notable difference in our data is that a far higher percentage of our students who could not explain a swap at week 3 could write a swap at week 5 (i.e. 71% cf. 57%).

Week 3, Explain a swap between two variables (see Figure 2)	Week 5, Successfully wrote an equivalent swap between two variables	
Wrong (n = 21)	71%	(cf. 57%)
Right (n = 30)	83%	(cf. 92%)
χ^2 test	p = 0.3	(cf. p = 0.001)

Table 2: Results from the replication at QUT, with comparative percentages shown in brackets from our original study (Corney, Lister and Teague, 2011).

Week 3, Explain a swap between two variables (see Figure 2)	Week 5, Successfully wrote a swap between two variables	
	failure	success
Wrong (n = 21)	6	15 (i.e. 71% of 21)
Right (n = 30)	5	25 (i.e. 83% of 30)

Table 3: The contingency table for calculating the chi-square value in Table 2.

2.1.1 Reflections

We suspect that the difference in our results is due to the way in which these tests were integrated with our teaching. In the original study, the week 3 test was not discussed with the class by the lecturer (i.e. co-author Corney). In contrast, our week 3 test was followed by a lengthy discussion of the test by the lecturer. (There was nothing pedagogically novel about that discussion. Corney discussed swapping in the same way a lecturer might discuss any piece of code.) Assuming our explanation for the difference is correct, our result may be encouraging, as it may indicate that the problems students face are amenable to pedagogical intervention. However, the question would still remain as to whether a student can transfer that learning to other programming problems. This is an issue to which we return in section 2.2.2.

In Table 2, we have presented our data in the same format we used in our original work. That format is an unusual format for presenting data that is then tested statistically by a chi-square test. Table 3 reproduces our data from Table 2 as the more traditional contingency table. In this paper, we will present our results in both forms.

2.2 Explaining a Sort of Three Variables

Another question in the week 5 test from our original study is shown in Figure 3. In that study, we reported a statistically significant result ($p < 0.05$) for student performance on this question in week 5 and the explanation question in week 3. Our replication results are shown in Tables 4 and 5. While our percentages in the replication are very similar to the percentages in our original study, our replication results do not quite meet the traditional 0.05 threshold of statistical significance, perhaps due to our smaller sample size.

Opinions vary on the interpretation of the 0.05 threshold for statistical significance. Some people view it as a rigid threshold – a result is either significant (i.e. below 0.05) or it is not significant. We are inclined to the alternative view, also commonly held, that the traditional 0.05 threshold is somewhat arbitrary (Cohen, 1994). The standard 0.05 threshold means that the chance of a data sample being a fluke is 1 in 20; whereas our 0.06 result simply means that the chance of our data sample being a statistical fluke is only slightly higher, at 1 in 17. We therefore argue that our replication results are weakly supportive of our original study. However, we also acknowledge it is possible that the effect we have observed in both the original study and this replication is on the margin of significance.

Alternately, one can argue that a p value around 0.05 is an encouragingly strong result, given that we are comparing student performance on just two explanation questions, one in week 3 and another in week 5. A more comprehensive test would involve asking several explanation questions in each of weeks 3 and 5.

2.2.1 Reflections

The results in Tables 4 and 5 support our suspicion that the earlier null result (i.e. Tables 2 and 3) may be due to how we taught the class. That is, even though the students may have rote learnt the swap code because of the emphasis we placed upon it in the lecture, the performance of students at explaining code in weeks 3 and 5 are consistent (albeit marginally consistent, at $p = 0.06$).

Even though the performance on the week 3 and week 5 explanation questions are (marginally) consistent, forty percent of students who answered well the week 3 explanation question did not answer well the week 5 explanation question. Such a backward step suggests (unsurprisingly) that some students who could reason correctly about the simpler code in week 3 could not transfer that reasoning to the week 5 problem containing an `if` statement.

Further to the point made in the previous paragraph, we wonder whether our use of a chi-square test is a pessimistic way of establishing the relationship between student performance on the week 3 and week 5 questions;

If you were asked to describe the purpose of the code below, a good answer would be “*It prints the smaller of the two values stored in the variables a and b*”.

```

if (a < b):
    print a
else:
    print b
    
```

In one sentence that you should write in the empty box below, describe the purpose of the following code.

Do **NOT** give a line-by-line description of what the code does. Instead, tell us the purpose of the code, like the purpose given for the code in the above example (i.e. “*It prints the smaller of the two values stored in the variables a and b*”).

Assume that the variables `y1`, `y2` and `y3` are all variables with integer values.

In each of the three boxes that contain sentences beginning with “Code to swap the values ...”, assume that appropriate code is provided instead of the box – do **NOT** write that code.

```

if (y1 < y2):
    Code to swap the values in y1
    and y2 goes here.
if (y2 < y3):
    Code to swap the values in y2
    and y3 goes here.
if (y1 < y2):
    Code to swap the values in y1
    and y2 goes here.
print y1
print y2
print y3
    
```

Sample answer: “it sorts the values so that $y1 > y2 > y3$ ”

Figure 3: A question from the week 5 test of Corney, Lister and Teague (2011).

Week 3, Explain a swap between two variables (see Figure 2)	Week 5, Successfully explained a sort of three variables (see Figure 3)
Wrong (n = 21)	33% (cf. 36%)
Right (n = 30)	60% (cf. 62%)
χ^2 test	$p = 0.06$ (cf. $p = 0.03$)

Table 4: Results from the replication at QUT, with comparative percentages shown in brackets from the original study.

Week 3, Explain a swap between two variables (see Figure 2)	Week 5, Successfully explained a sort of three variables (see Figure 3)	
	failure	Success
Wrong (n = 21)	14	7 (i.e. 33% of 21)
Right (n = 30)	12	18 (i.e. 60% of 30)

Table 5: The contingency table for calculating our chi-square value in Table 4.

for example, consider Table 5. A chi-square test focuses on consistency – whether most students answer both questions incorrectly (i.e. 14 in Table 5) or correctly (i.e. 18). It is not contrary to our argument, however, that some students would answer the week 3 question correctly, but the week 5 question incorrectly (i.e. 12). Our argument is merely that, in the absence of a pedagogical intervention, students who answer the week 3 question incorrectly will tend not to answer the week 5 question correctly (i.e. 7, which is 33% of 21)

2.2.2 Write the Swap but Explain the Code

To further test the idea that students had rote learnt the swap code in week 5, we looked at the $n = 40$ students who wrote the swap code correctly in week 5, and considered how well those students did on the week 3 and week 5 explanation tasks. The results are shown in Tables 6 and 7. These large differences in the two percentages (33% vs. 64%) do suggest that students who struggled to explain previously unseen code in week 3 tended to continue to struggle to explain previously unseen code in week 5. However, these percentages are not conclusive, as a χ^2 test produces a p value that is just over the traditional 0.05 threshold of statistical significance. We suspect that, with a slightly larger sample, the p value would meet the traditional 0.05 criterion.

Earlier, we suggested that the results in Tables 2 and 3 may be pedagogically encouraging, as those results may indicate that the problems students face are amenable to pedagogical intervention. In contrast, the results in Tables 6 and 7 are pedagogically discouraging – while students may have rote learnt how to swap the values of two variables, those students did not then manifest a strong ability to answer the week 5 explanation question.

3 Replication at UTS

In the replication at UTS, students were tested a little later in semester, at weeks 5 and 7. This was because the students were being taught an objects-early introduction to Java, so some of the concepts in the two tests were taught a little later in the semester.

In the week 5 test we used slightly different tracing questions to screen the students, but our questions also only involved assignment statements, and we do not regard these questions as being significantly different. After screening, 64 students remained.

We made one change in the replication that is arguably non-trivial. We changed one of the week 3 questions

from the version shown in Figure 2 to the version shown in Figure 4.

Week 3, explain a swap – for the $n=40$ who wrote a correct swap in week 5	Week 5, explain a sort of three variables
Wrong (n = 15)	33% right
Right (n = 25)	64% right
χ^2 test	$p = 0.06$

Table 6: Performance of the 40 students who wrote a correct swap in week 5, on the week 3 and week 5 explanation problems.

Week 3, explain a swap – for the $n=40$ who wrote a correct swap in week 5	Week 5, explain a sort of three variables	
	failure	success
Wrong (n = 15)	10	5 (33% of 15)
Right (n = 25)	9	16 (64% of 25)

Table 7: The contingency table for calculating the chi-square value in Table 6.

<p>The purpose of the following three lines of code is to swap the values in variables a and b, for any set of possible initial integer values stored in those variables:</p> <pre>c = a; a = b; b = c;</pre> <p>In one sentence that you should write in the box below, describe the purpose of the following three lines of code, for any set of possible initial integer values stored in those variables:</p> <pre>j = i; i = k; k = j;</pre> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p><i>Sample answer:</i> “it swaps the values in i and k”</p> </div>

Figure 4: The modified question used in the replication at UTS. The original form of the question is in Figure 2.

3.1 Writing a Swap

Table 8 summarises our results from this part of the replication, where we consider student performance on explaining the swap code at week 5 and writing swap code at week 7. These results in Table 8 do not support our original results. As for the QUT replication, we suspect this null result is due to the week 5 test being followed by a lengthy discussion of the swap code by the lecturer (i.e. co-author Lister).

Week 5, Explain a swap between two variables (see Figure 4)	Week 7, Successfully wrote an equivalent swap between two variables
Wrong (n = 11)	82% (cf. 57%)
Right (n = 30)	79% (cf. 92%)
χ^2 test	p = 0.8 (cf. p = 0.001)

Table 8: Results from the replication at UTS, with comparative percentages shown in brackets from the original study by Corney, Lister and Teague (2011).

3.2 Explaining a Sort of Three Variables

In our original study, we reported a statistically significant result for student performance on explaining swapping in week 3 and explaining the sorting of three variables in week 5. The results from our replication are shown in Tables 9 and 10. Our results emphatically confirm the results of the original study. As with the equivalent results from the replication at QUT, these results support our suspicion that the earlier null result (i.e. Table 8) is due to the lengthy lecture discussion about the swap code that followed the week 5 test.

Week 5, Explain a swap between two variables (see Figure 4)	Week 7, Successfully explained a sort of three variables (see Figure 3)
Wrong (n = 11)	27% (cf. 36%)
Right (n = 53)	91% (cf. 62%)
χ^2 test	p < 0.001

Table 9: Results from the replication at UTS, with comparative percentages shown in brackets from our original study (Corney, Lister and Teague, 2011).

Week 5, explain a swap between two variables (see Figure 6)	Week 7, explain a sort of three variables (see Figure 5)	
	failure	success
Wrong (n = 11)	8	3 (i.e. 27% of 11)
Right (n = 53)	5	48 (i.e. 91% of 53)

Table 10: The contingency table for calculating the chi-square value in Table 9.

4 Reflection: Ambiguity in Natural Language

A common concern among academics about “Explain in Plain English” questions is the possibility of ambiguity in student responses (e.g. Simon and Snowdon, 2011).

We found little ambiguity in our student responses – most answers were clearly right or wrong. For example, for the question on swapping values of two variables shown in Figure 2, some student responses that we judged as correct are:

swap b and c
swap contents of b and c using a as temp
swap values of c and b; leaving original value of b in a

Figure 5 shows some wrong answers given by students for this question. Most of these answers are clearly wrong.

For the question on sorting the values of three variables shown in Figure 3, some student responses that we judged as correct are:

orders in descending
places in descending
prints in order of highest to lowest
reorders in descending
sorts from largest to smallest
sorts in descending
swaps into descending

Figure 6 shows some wrong answers given by students for this question. Again, most of these answers are clearly wrong. Our experience is that grading student responses to explain in plain English questions is straightforward – arguably more straightforward than reading the confused code that students often write in exams.

- 1) a b and c have the same value
- 2) assigns a to b, b to c, c to b. No overwriting
- 3) b overwrites a; c overwrites b; then a overwrites c. b ends up in c
- 4) in the end, a will equal c, and c will equal a, both a and c hold same values
- 5) replaces c with b
- 6) sets a b and c to the value of b
- 7) swap values in b and a
- 8) to change every variable's value to that of b

Figure 5: A selection of wrong answers given by students at QUT for the code that swaps the values between two variables (see Figure 2).

- 1) assigns y3 the smallest value, y2 winds up with the default value
- 2) determines if a value is lower than another, then prints them all
- 3) if variables are smaller it will swap them to be larger at the end
- 4) printing, swapping y1 and y3 if y1 smaller
- 5) prints larger of 2 variables
- 6) prints largest value if one variable is smaller than the other
- 7) removes lowest and replaces with a value higher than it originally had
- 8) swap y1 and y3
- 9) swaps and prints
- 10) swaps codes for smaller value then print
- 11) swaps values y1 and y3, but y2 remains the same

Figure 6: A selection of wrong answers given by students at QUT for the Figure 3 code that sorts three variables.

5 Extension: A Third in-Class Test

At QUT, we went beyond the original study (Corney, Lister and Teague, 2011), but in a fashion very much in the same style as the original study, by conducting a third test in week 7 (i.e. mid-semester). In this week 7 test, we asked the students to write code to sort the values in an array with three elements. The code they needed to write is the same, algorithmically, as code they were asked to explain in week 5 (see Figure 3). However, the code they had to write in week 7 was not identical to the week 5 code, for the following reasons:

- Whereas the code in week 5 used three separate variables, the code in week 7 used a list of three elements.
- Students were required to write the actual assignment statements to swap values among the variables.

There were 48 students who did both the week 5 and week 7 tests. The results for all 48 students are shown in Table 11. It is not surprising that a low percentage (12%) of students who could not explain the code in week 5 could not also write similar code in week 7. More surprising was that only 35% of students who could explain the code in week 5 could write similar code in week 7. Once again, our $p = 0.06$ is just above the traditional 0.05 threshold for statistical significance, but as before we are inclined to believe that our results are weakly consistent with our original study, without meeting the traditional 0.05 threshold.

Week 5, explain a sort of three variables (see Figure 5)	Week 7, successfully wrote a correct sort of an array with 3 elements
Wrong (n = 25)	12%
Right (n = 23)	35%
χ^2 test	$p = 0.06$

Table 11: Results from week 7 test at QUT.

6 Replication: End of Semester Exam

In our original study, we reported a statistically significant relationship between student performance on their in-class tests and a code writing task in the final exam. In this section, we report our replication, again carried out at QUT.

In this replication, the code writing question in our end of semester exam was not the same as the question used in our original study. Our question in the replication required students to write code to move the elements of an array one place to the left, wrapping the leftmost element around to the rightmost position. One possible solution to this question is shown in Figure 7.

We screened students, using two tracing questions from the week 7 test. Both of these screening questions required students to trace iterative code operating on an array. If a student answered correctly at least one of those two questions, the student was judged as having demonstrated (as early as week 7) an understanding of the semantics of loops and lists. Since tracing iterative code is an error prone activity, especially as early as week 7, we felt that success on one question was sufficient evidence of understanding. Furthermore, tracing code

with 50% accuracy is consistent with Lister's (2011) definition of the pre-operational stage in the novice programmer.

```
temp = x[0]
for i in range(0, len(x)-1, 1)
    x[i] = x[i+1]
x[len(x)-1] = temp
```

Figure 7: A solution, in Python, to the code writing question in the final exam.

When writing the solution to the problem in Figure 7, a student must provide a suitable assignment for the loop body, either $x[i] = x[i+1]$ as shown in Figure 7, or $x[i-1] = x[i]$. We feel that students who failed to provide such an assignment statement demonstrated a profound misunderstanding of the question (perhaps due to English being their second language), so we also eliminated from our analysis any student who did not provide one of those two suitable assignment statements. The screening left us with a sample of 40 students.

Since this paper has emphasised the concept of swapping, our analysis of this exam question focuses upon the swapping component in this final exam question, especially the first and last lines as shown in Figure 7. The first line saves the leftmost element to a temporary variable, and the fourth line copies that temporary value back into the array.

(We note in passing that few students in the class gave a completely correct solution to this code writing problem. The most common errors in near-correct solutions were off-by-one errors in the loop. Often, the values through which the control loop variable "i" would iterate were appropriate, in isolation, and so was the assignment statement in the body of the loop. However, those two lines, in combination, were often not compatible.)

Table 12 breaks down the performance of students on this code writing task from Figure 7, according to whether the students were able to explain similar code in the week 7 test. Among students who could not explain that code in week 7, only 42% correctly handled the end element in the final exam, compared to 86% of students who did explain that code in week 7. A χ^2 test produces a statistically significant p value. Our result is therefore strongly supportive of our original findings.

Week 7, explained a shift (see code in Figure 3)	End of semester exam, write code to shift elements in an array (see Figure 7), correct treatment of the end element in lines 1 and 4
Wrong (n = 26)	42%
Right (n = 14)	86%
χ^2 test	$p < 0.01$

Table 12: Relative performance on the explanation task in week 7 and writing similar code in the final exam, at institution A (n=40).

7 Conclusion

Our empirical results support our original findings, with the following caveats.

In replications at both of our institutions, we did not find a relationship between students being able to explain swap code and being able to write similar code two weeks later. We believe this failure was because our teachers talked about the swap code between the two tests. In general, we think the relationship between explaining code and writing code found in our original study will only occur when there is not a pedagogical intervention between the two tests.

Some of our results were just outside the traditional 0.05 boundary of statistical significance, at $p = 0.06$. How readers will regard those results depends upon their view of the traditional 0.05 boundary. Some readers will maintain that a result is either significant (i.e. $p < 0.05$) or it is not significant. As we have argued earlier in the paper, we are inclined to the alternative view, which we believe is more statistically sophisticated, that the standard 0.05 threshold means that the chance of a data sample being a fluke is 1 in 20; whereas our 0.06 result simply means that the chance of our data sample being a statistical fluke is only slightly higher, at 1 in 17. We therefore argue that those replication results with $p = 0.06$ are weakly supportive of our original study, while acknowledging that our results do not meet the traditional $p = 0.05$ criterion. However, it is also possible that the effects we have reported in both the original study and in these replications are on the margin of significance. Further replication work, at other institutions, is warranted. Especially interesting would be further replication work that uses more than a single explanation question in each of the two weeks under comparison, as using only a single explanation question in each week may be the source of the statistical uncertainty.

One of our empirical results strongly supports our earlier findings, without the need for any caveats – we found that students who could not demonstrate an ability to explain a piece of code in week 7 of semester tended to do more poorly at attempting to write similar code at the end of semester.

Overall, this replication study and its minor extensions has increased our confidence in the conclusions we drew in the original study – the problems some students face in learning to program are not due to the more complex programming constructs they are taught in the second half of semester, but instead begin in the first half of semester.

References

- Cohen, J. (1994) The Earth is Round ($p < .05$) *American Psychologist*, 49(12). pp 997-1003.
- Corney, M., Lister, R., and Teague, D. (2011) *Early Relational Reasoning and the Novice Programmer: Swapping as the "Hello World" of Relational Reasoning*. Thirteenth Australasian Computing Education Conference (ACE 2011), Perth, Australia, January 2011. pp. 95-104.
- Lister, R., Fidge C. and Teague, D. (2009) *Further Evidence of a Relationship between Explaining, Tracing and Writing Skills in Introductory Programming*. Fourteenth Annual Conference on Innovation and Technology in Computer Science Education, Paris, France. pp. 161-165
- Lister, R., Clear, T., Simon, Bouvier, D., Carter, P., Eckerdal, A., Jackova, J., Lopez, M., McCartney, R., Robbins, P., Seppala, O., and Thompson, E. (2010) Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *SIGCSE Bull.* 41, 4 (January), 156-173.
- Lister, R. (2011). *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. Thirteenth Australasian Computing Education Conference (ACE 2011), Perth, Australia. CRPIT, 114. John Hamer and Michael de Raadt Eds., ACS. 9-18.
- Lopez, M., Whalley, J., Robbins, P., and Lister, R. (2008) *Relationships between reading, tracing and writing skills in introductory programming*. Fourth International Workshop on Computing Education Research, Sydney, Australia, 101–112.
- Simon and Snowdon, S. (2011) *Explaining program code: giving students the answer helps – but only just*. Seventh International Computing Education Research Workshop (ICER), Providence, Rhode Island, pp. 93-99.
- Venables, A., Tan, G. and Lister, R. (2009) *A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer*. Fifth International Computing Education Research Workshop (ICER), Berkeley, CA. pp. 117-128.

Models and Methods for Computing Education Research

Mats Daniels and Arnold Pears

Department of Information Technology
Uppsala University
PO Box 327, 751 05 Uppsala, Sweden

mats.daniels@it.uu.se, arnold.pears@it.uu.se

Abstract

We have been engaged in computing education research for close to two decades. One characteristic of the field has been a preponderance of exploratory research, Marco Polo papers as Valentine termed them. Even considering the entire research corpus it is hard to discern a clear trend in terms of models and methods for conducting research. While some prominent researchers, such as Fincher, have established a tradition of mixed method research and multi-institutional studies, these approaches form a branch of the discipline and do not constitute a dominant paradigm. Indeed computing education research demonstrates an observable eclecticism in relation to method, combining as it does approaches from a range of qualitative and quantitative research traditions. A consequence of this is that we have spent time on thinking about the research area as a whole. We believe that a key defining feature of computing education research is the focus on learning in the discipline. The point of departure for much computing education research is consequently a need to address educational challenges in the discipline, rather than a standpoint in an educational tradition. This places the research objective, or question, in focus and makes the choice of method a secondary concern for many computing education researchers. In this article we discuss the nature of a broader emerging paradigm for conducting educational research, and a framework which can scaffold working within this paradigm.

Keywords: Paradigm, Educational Research, research Framework

1 Introduction

Since much of computing education research is driven by a pragmatic goal to understand learning phenomena associated with complex disciplinary knowledge/concepts it can be hard to associate the resulting scholarly output with an established research paradigm. In fact we propose that a certain degree of methodological eclecticism may be inherent in the practice of research, and scholarly practice in teaching and learning in our discipline.

This derives from the fact that the framing of research questions is based on a desire to better understand learning in a context, thus the choice of method often depends on the type of insight deemed most useful in that particular context. As a consequence it is not unusual for a single researcher to conduct both qualitative and quantitative studies, and subsequently drawing on elements of action research, phenomenography and statistical analysis to substantiate claims.

We have pursued the following question

How can research-based computing educational development be structured?

during the past two decades and have in that pursuit built a general education research foundation to complement our competence in the computing area. We will in this paper first reason generally about computing education research in terms such as its context, how it could be conducted, and the philosophy behind it. This is followed by presenting a framework for research in computing education and for conducting development in an action research manner. We will conclude with a case study based on using action research.

Our work can be seen as developing a paradigm for conducting educational research and our framework can be seen as illustrating how to scaffold working according to such a paradigm.

2 Computing Education Research in Context

Computing education research provides a bridge between education research and computer science research, contextualizing educational research to help to facilitate student learning of computer science knowledge, research concepts and general theory. We argue that disciplinary educational research provides insight into the application of general educational theory to learning in a specific domain. The combination of disciplinary depth in both a scientific field and education research allows researchers to identify and investigate teaching and learning challenges in a way that is richer in terms of its disciplinary content.

Education at research universities is characterized by research informed teaching, where high level research permeates the educational environment. In this context high performance learning depends on research in both the scientific subject matter itself and in the learning of advanced computing concepts. Educational research alone is not sufficiently accessible to many science researchers, which presents challenges in terms of adapting new educational models to teaching. Likewise, educational researchers often have little exposure to advanced research concepts in computing, which affects

their ability to discern and study learning phenomena related to tertiary teaching and learning challenges.

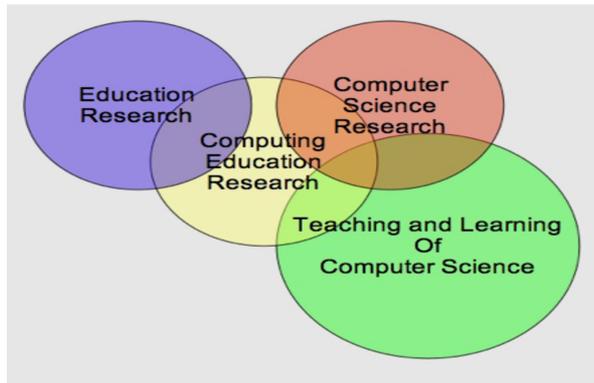


Figure 1: Contextualization of computing education research

The focus of disciplinary educational research on the instructional challenges of the discipline predispose researchers to adopt a very pragmatic stance in terms of methods for data collection and analysis. In this sense we believe that computing education research differentiates itself from general educational research. Rather than taking its point of departure in a research paradigm or tradition, the point of departure in computing education research is in the question and the type of answers to the question that appear to the researcher to be most relevant when addressing the educational challenge the question represents.

While this approach is powerful in its pragmatism it also presents challenges to the discipline. Methodological diversity can be construed as undermining the reliability and coherence of the research discourse. We argue that this is not necessarily the case, especially if computing education research adopts a broader methodological framework which allows question, method and analysis to be presented in a well argued manner.

3 Theoretical Foundation

It is vital to establish a theoretical foundation for work to be presented in order to provide the reader with insights into how the research has been conducted and the scope and generalizability of the results. The theoretical foundation can be put in a holistic perspective by the research framework we will present and the action research methodology (Lewin 1946) can be used to provide means to reason about choices of research methods and the nature of results on how to address learning issues.

The structure of a research ecology is discussed in some depth by Crotty in the introduction to his book "The Foundations of Social Research" (Crotty 1998). He uses the following image to depict the relationship between the four terms *epistemology*, *theoretical perspective*, *methodology*, and *method*.

The relationship presented in figure 2 can be described as follows: The *epistemology* is more or less a fundamental part of the particular researcher conducting a study and it is strongly connected to the *theoretical perspective* the researcher is applying in the study. The

theoretical perspective has implications for which *methodologies* that are suitable. The particular *method* associated with the methodology selected in the study is applied according to the theoretical perspective underpinning the study.

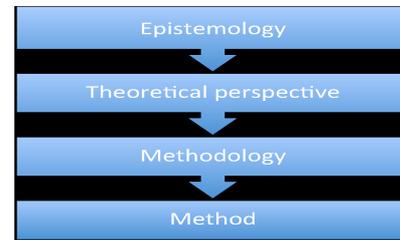


Figure 2: A research ecology (adapted from Crotty 1998, p. 4)

Below, a more detailed description of these terms, as used in this paper, is given before entering into a more detailed discussion of the research framework we have developed and how we use the action research methodology.

3.1 Epistemology

An epistemology is the theory of knowledge embedded in the theoretical perspective and thereby in the methodology. Objectivism, constructivism, and subjectivism are examples of epistemologies. A theoretical perspective involves knowledge and the epistemology deals with understanding what knowledge is, how we know what we know, or to quote Maynard (1994):

Epistemology is concerned with providing a philosophical grounding for deciding what kinds of knowledge are possible and how we can ensure that they are both adequate and legitimate. (p. 10)

3.2 Theoretical Perspective

A theoretical perspective is the philosophical stance underlying the methodology and thus providing a context for the process and grounding its logic and criteria. Positivism, symbolic interpretivism, hermeneutics, and critical inquiry are examples of theoretical perspectives. By stating the theoretical perspective used a reader can gain an understanding of the assumptions, the way of looking at the world and making sense of it that guided the choice of methodology.

3.3 Methodology

Methodology can be seen as the strategy, the plan of action, process or design lying behind the choice and use of particular methods and linking a choice and use of methods to the desired outcomes. Experimental research, ethnography, grounded theory, action research, and discourse analysis are examples of methodologies. In research one should not just name and possibly describe the methodology selected, but also account for the rationale it provides for the choice of methods and the way the methods are used.

3.4 Methods

Methods are the techniques or procedures used to gather and analyze data related to some research question or hypothesis. Sampling, questionnaire, participant observation, interview, focus group, case study, narrative, statistical analysis, interpretative methods, and content analysis are examples of methods. It is important to be specific in describing how a method is used, e.g. stating what interview technique is used, and in what setting, instead of just describing it as carrying out interviews.

4 A Framework for Educational Research and Development

Educational research results stem from a wide range of different research traditions. Computing educators are often unfamiliar with the kind of results educational research produces and these results can be non-trivial to use as a basis for development. The difficulties originate from educators having specific questions related to a particular course unit or to general issues regarding some particular aspects of the computing or engineering domains, whereas educational research results often are at an abstract level regarding learning in general. Practical models with which to pursue research-based development of computing education are needed as a result.

There are also issues to consider when computing educators conduct educational research. One example, from our experience in reading the literature, is that they seldom document the learning environment and especially not the context in which it exists. This might be due to space limitations on conference papers, but could also depend on the authors being too focused on their own learning environment. Neglecting to do this reduces the trustworthiness and usefulness of the research results.

The questions of interest to computing educators are mostly related to the development of a course unit, both in terms of how to construct a learning environment and understanding what is happening during, or after, an instance of a course unit. The ways to find answers to these types of questions vary, but are often based on using qualitative methods (Berglund et al. 2006).

In order to understand and evaluate results it is important to know which research methods were used, which research methodologies they belong to, and the epistemology and theoretical perspective that underpins the study. This section is based on early work on defining a framework for how to conduct computing education research (Pears et al. 2002, Pears and Daniels 2003). That there is a place for such a framework can be deduced from this statement by Crotty (1998):

Research students and fledging researchers – and, yes, even more seasoned campaigners – often express bewilderment at the array of methodologies and methods laid out before their gaze. These methodologies and methods are not usually laid out in highly organized fashion and may appear more as a maze than as pathways to orderly research. There is much talk of their philosophical underpinnings, but how the methodologies and methods relate to more theoretical elements is often left unclear.

To add to the confusion, the terminology is far from consistent in research literature and social science texts. One frequently finds the same term used in a number of different, sometimes even contradictory, ways. (p. 1)

4.1 Learning environment

The context of a research question is an essential part in understanding results for a broader community than the local colleagues. The context includes, for instance, the degree program in which a course unit exists and the formal specification of the course unit, e.g. learning objectives and content. The students taking the course unit and especially the educators responsible for an instance of a course unit also constitute part of the learning environment.

The influences the educators bring to the learning environment are both explicit, for instance the selection of examination methods and tools provided, and implicit in the influence of their epistemology regarding learning and knowledge. Tools are to be understood as representing anything that is brought in to the learning environment to aid the students' learning, and the range of what is considered a tool is almost limitless, examples being assignments, books, clickers, labs, quizzes, and web-based self-study material. The importance in capturing the epistemological view derive from that it may influence how much students are encouraged to be active in their learning and also what constitutes learning in the view of the educator(s).

The research questions can range from concrete aspects of a particular course unit to general educational issues, e.g. in computing education how to establish a learning environment for novices learning to program. Another example is questions related to aspects of using open problems in a computing learning environment. These questions are better understood if a reader has a clear view of the intended learning environment.

A visual representation of the context influencing the development of a research question, i.e. the external scope, is given in figure 3:

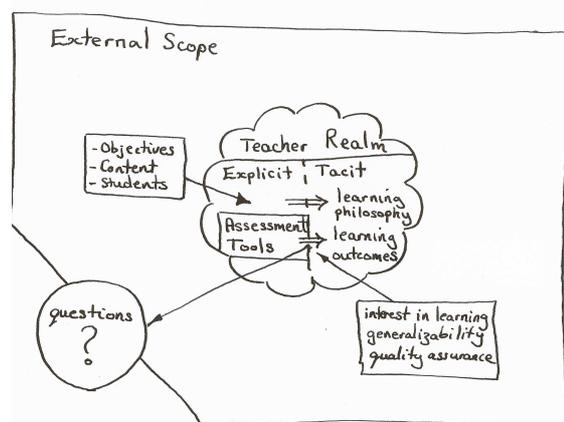


Figure 3: The learning environment for the research question

Figure 3 is part of a graphical approach to describing the context and influences that have a bearing on the development and conduct of educational research. This figure provides a detailed view of one aspect of the more

general framework presented in figure 4, that has grown out of discussions in Uppsala Computing Education Research Group (UpCERG).

Figure 3 is intended to capture the relation between the overall learning environment, especially how it is viewed by the educator (or educators) involved, and the research question. The researcher is reminded to consider and explicitly document the external scope in terms of for instance:

- Formal specifications of learning objectives for the course unit.
- Educational context in the form of degree program.
- Information about the students attending the unit.
- General issues related to the research question such as the educators:
 - Interest in learning.
 - Desire to find transferable answers.
 - Striving for quality assurance.

An important objective is to capture issues with respect to the educators involved:

- Explicit choices such as the most appropriate means of assessing students and the available educational tools.
- Tacit influences, such as epistemology and their view on what constitutes learning.

4.2 Research Setting

Capturing the relevant aspect of the learning environment is an important step in the process of developing research questions. The next step is to find a suitable method with which to find an answer to the formulated question. There is no underlying assumption in terms of epistemology or theoretical perspective in the research framework, nor on which research methodology to base the use of the selected methods on. The framework is intended to support the researcher in selecting methods and documenting the theoretical rationale for the choice. That is, the framework should be used to provide the researcher with a clear connection between the aspect of the research question addressed by the chosen research method and associated research methodology and the assumed theoretical base, i.e. epistemology and theoretical perspective, for the answers provided.

Making well-informed choices of which method to use is often beyond an individual computing educator wishing to conduct a research study and the communication with scholars from other disciplines to learn more about the available methods might be problematic. This problem is, in our experience, to a large extent based on not sharing a common research terminology, nor having the same research interests. The framework is intended to support both making the choice and facilitating communication, by providing a base to place the question and scaffold thinking about where to find ways to reason about the question and the limits and possibilities of different approaches to investigating the question.

The epistemology and theoretical perspective are associated with the person who formulated the question, although it is of course possible for a person to choose between different theoretical perspectives depending on

which aspect of a research question they might wish to address. The choice of epistemology and theoretical perspective is not part of this framework, but we have introduced choice of discipline as a level in the framework. This is done in order to get a frame of mind about where to find suitable research methodologies and methods, e.g. that different disciplines within social sciences might be a good place to start if one wants to find out something about cultural influences in a learning environment.

The next step is to find a suitable research methodology that has promise with regard to the question. The discipline lens might be useful in finding this, perhaps through interaction with researchers in that discipline. The first steps in the process, i.e. to capture the relevant aspects of the learning environment, phrasing the research question, and selecting the potential discipline to aid in finding an answer, provides the start for creating a common ground between the computing educator(s) formulating the question and the researchers in the selected discipline(s). This could typically lead to changes in how the learning environment is viewed, e.g. that more aspects should be documented.

In the framework we depict computing education research (CER) as the outermost layer, in which the studies based on the chosen research methods are performed. It is here that the questions are answered.

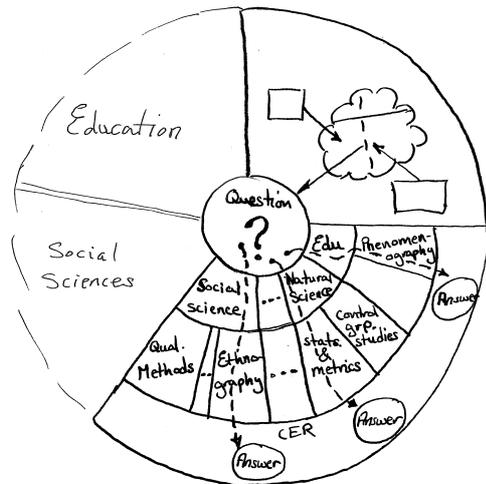


Figure 4: The educational research framework

An objective of this framework is to raise the level of scholarliness among educators and educational researchers in the computing discipline. The idea is to provide a structure for integrating development and research and aid in capturing the relevant issues that will make development and research efforts more transferable. The work reported on in this paper, apart from presenting the framework as a result, is an example of the influence arising from this general framework for the work of a computing education researcher. This is done by illustrating how the framework provides a context for addressing learning environment questions based on a variety of learning theories, as well as setting the stage for working in an action research manner.

4.2.1 A Course Unit Perspective

Many questions stem from the context of a course unit. Figure 5 illustrates a view that is derived from the framework intended to capture some of the issues and actions that relate to conducting discipline education research.

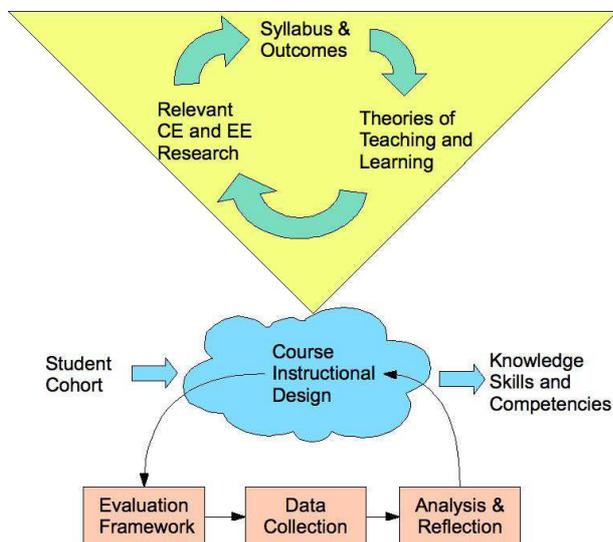


Figure 5: Course unit centered research elements

The center of the illustration is the actual course unit, with its influx of students taking the course unit and the knowledge, skills and competencies that are supposed to be developed by the incoming student cohort.

The triangle contains aspects of research questions that typically surfaces when dealing with issues related to a course unit. The syllabus and outcomes are either the external limits which a researcher has to adhere to, or which to change. It is essential to base the question on theories of teaching and learning, or in some cases it might be that such theories are developed.

The lower part of figure 5 illustrate the type of research that is done to investigate the question at hand. This includes deciding on an evaluation framework that will give a setting in which the question can be addressed, the means with which to collect data, and not least how to perform an analysis of the collected data and reflect on the results.

The research context captured by the design illustrated in figure 5 fit well with conducting a cycle in an action research study. A short presentation of action research and a case study is given below.

5 Action Research

The term *action research* is attributed to Kurt Lewin at MIT, who used it in his paper “Action research and minority problems” (Lewin 1946). He described the methodology as comparative research on the conditions and effects of various forms of social action and research leading to social action that uses a spiral of steps, each of which is composed of a circle of planning, action, and fact-finding about the result of the action, or in other words experimenting by making changes and

simultaneously studying the results, in a cyclic process of planning, action, and fact gathering. Lewin had a strong positivist view and our example is based on a constructivistic view. Action research is thus an excellent example of a research methodology that is connected to different theoretical perspectives.

Action research includes a strong relationship between the researcher(s) and the practitioner(s) and an open attitude to which data collection methods to use (Rasmussen 2004, Reason 2006, McKay and Marshall 2001). The essence of action research is well captured by Carr and Kemmis (1983) who state that an action research activity has two essential aims, i.e. to *improve* and to *involve*, and that the focus of the improvement lies in three key areas: improving a practice; improving the understanding of a practice, and improving the situation in which the practice takes place.

The rather open description of action research lends itself to different interpretations. Approaches to action research are widely discussed in the literature, e.g. (Reason and Bradbury 2007, Elden and Chisholm 1993, Cajander 2010), where it is pointed out that there is a common core that has been adapted to different contexts. The way action research is carried out is heavily influenced by the specific problem addressed, the relationship between the researcher(s) and practitioner(s), and the discipline within which the research is situated.

The role of the researcher in action research is also a topic of discussion. Extreme positions on the role of the researcher include a focus on the research aspect and data gathering, almost to the point of being a spectator in the process, or a focus on the service aspect by fully collaborating with the practitioners in solving the problem (Westlander 2006). In practice a situated approach which is a mixture of the two poles is often used, typically due to the complexity and situated nature of the problems addressed (Cajander 2010).

A duality of the role of the researcher is discussed by McKay and Marshall using a model with two different cycles; an explicit problem solving cycle and a research cycle (McKay and Marshall 2001). McKay and Marshall also emphasize another aspect of action research; that one result of working in this manner can be seen as developing a theory around the issue addressed.

The role of the practitioners in action research is also discussed in the literature (Elden and Chisholm 1993), with a growing interest in considering practitioners as peers in the research process. Examples of practitioners are students, clients, educators and other experts who contribute with their knowledge and understanding. The extent of involvement typically varies depending on the problem addressed.

5.1 Action Research in the IT in Society Course Unit

An illustration of the steps within a single action research cycle in the context of developing the IT in Society course unit (Laxer et al. 2008) is given in figure 6. This course is based on the Open-Ended Group Project concept (OEGP) (Daniels 2011) and has among its aims to develop the students’ inter-cultural competence.

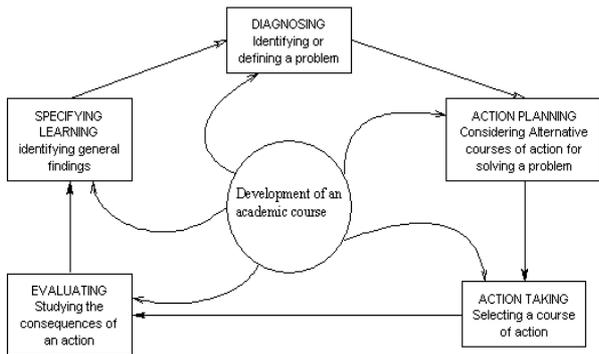


Figure 6: The Action Research Cycle (adapted from Suman and Evered 1978)

A starting point for a description of the action research cycle can be the top box, where identification and an initial analysis of the specific problem to be addressed are done. The next box in the cycle represents the process of preparing for setting up an action plan addressing the identified problem. This involves, apart from describing different alternative actions, documenting the theoretical underpinnings for selecting an action. The “action taking” box represents the selection process, where the alternatives are compared in order to find the most appropriate action for addressing the identified problem. This process also involves reasoning about the methods to be used in evaluating the outcome of the action. The next step is to carry out the selected action plan, including gathering and analyzing data generated from the chosen research method. The last box before returning to the starting point represents abstracting answers relative to the identified problem, answers that will be used in starting the next cycle by looking at the problem with the added information from the action research cycle at hand.

Taking a lap around the action research cycle has some clear connections to activities described in the research framework. For instance, the starting point can be seen as selecting the research question: selection of research methods and documenting the theoretical underpinnings is an activity that is made easier by the research framework. Making answers more transferable typically involves anchoring them in a theoretical context and this is an activity that is facilitated by the research framework.

This model describes a rational and systematic inquiry action research, however, we concur with Reason (2006) who argues that these cycles are slightly “messier” than the neat diagrams drawn. Our own research has typically elements of being more diffuse and tacit as described by Reason (2006), even though the academic year provides a natural planning window, e.g. in the case of the development of the IT in Society course unit, for an action research cycle.

The academic year cycle provides an opportunity for reflection, taking stock of the progress made and learning gained in the previous cycle and serving as a logical planning point for the subsequent cycle. Outcomes and observations arising from an action plan for the current course instance naturally feed through into the design of the next.

The areas of the course unit addressed in the action plan for the following course instance might by different,

at least partially, from those addressed in the current (and previous) instance(s). There might also be a difference between cycles due to changes in the pedagogical and conceptual framework between consecutive course instances. These changes are an integral part of the analysis for each action cycle.

Five elements are emphasized within an action research framework inspired by McKay and Marshall (2001), which enable a conscious separation of the practice components from the research elements. They point out that this enables the researcher to avoid a common trap in action research: having the work described as “consultancy”. That is, they worry about not being taken seriously and argue that using their action research framework to anchor the answering of research questions in an applicable theoretical context provides a “visible” rigor to action research and thus address the issue of not being taken seriously..

The five elements are:

- **F**, the research framework or conceptual element informing the research, which in the terms used in this paper correspond to epistemology, theoretical perspective and concepts underpinning the research;
- **M_R**, the research methodology to be adopted;
- **M_{PS}**, the problem solving method that will be used in the practice situation;
- **A**, the problem situation of interest to the researcher (the research questions);
- **P**, the problem situation in which we are intervening (the practice questions of interest to the practitioners).

Examples of application of this action research framework to the work on the IT in Society course unit is presented in table 1 by giving an overview of different issues and approaches used to develop the course unit over the years.

This cyclical pattern of action-research-based development produces a progressive improvement of the theoretical base for creating a learning environment suitable for the selected learning outcomes.

This example can be complemented by viewing it according to Figure 5. Such a view results in a more concrete caption on one cycle in the action research process. This can be illustrated by the introduction of an expert on cultural awareness as an intervention in the course unit. This intervention was thus our instructional design as depicted by the cloud in Figure 5.

The intervention was based on us identifying that we wanted to address the learning outcome *to be able to evaluate and analyze one’s abilities and competencies regarding working in a multi cultural project*. Constructivism (Piaget 1970) was identified as a suitable theory to the view of learning associated with the intervention. We identified trust to be an important factor in collaboration based on educational research studies (Jarvenpaa et al. 1998, Panteli and Duncan 2004, Coppola et al. 2004).

Based on the identified intervention and the theoretical foundation captured by the triangle part of Figure 5 we designed the components of the study needed to evaluate the intervention. Part of the evaluation framework box

was to identify a suitable definition of intercultural competence (Byram, Nichols, and Stevens 2001), since it was important in understanding what it was to be learnt. The method for data collection was asking the students to reflect on the value of the seminar with the cultural awareness expert and this was followed by the researchers analyzing the reflections and themselves reflecting on the learning outcome. Further reading relative this example is given in Daniels’ thesis (2011).

Element	Description
F (Framework)	Constructivism, the OEGP concept, threshold concepts, conceptual change, communities of practice, cognitive load, collaborative technology fit, etc.
M_R (Research Methodology)	Action Research
M_{PS} (Problem solving method)	ITiS course unit and task design, international collaborations, local sponsor, reflective practitioner model
A (problem situation of interest to the researcher)	How does OEGP support or hinder the work of global student teams? How does OEGP develop student skills in global collaboration? How does OEGP develop each student’s professional skills and ability to cope with ambiguity and complexity, and to take responsibility for his/her own learning?
P (a problem situation in which we are intervening)	Improving teaching and learning through active learning approaches Students as active co-researchers Collaborative learning models Developing student capabilities in teamwork, cross cultural communication and use of IT Providing an interesting and meaningful learning experience Improving viability of student teams engaged in international teamwork

Table 1: Examples of elements of research investigating the IT in Society course unit

6 Conclusions

This paper discusses the nature of the computing education research (CER), arguing that the difference between CER research studies and those more prevalent in education research lies in the point of departure, or focus of the research. CER addresses concrete teaching and learning challenges in the discipline drawing on those methods appropriate to the context of the question being investigated,

We argue that it is this pragmatic focus on the question as paramount, that characterizes CER and other discipline based education research. The question, and the nature of useful answers, dictate the choice of methods for data collection and analysis to a much greater extent than is normal in education research. As a result there is a need for a framework which assists researchers in contextualizing their study, and describing the context at a level of detail that permits generalization.

In this paper we have described a framework that we believe is useful as a guide in describing the critical features of pragmatic research in CER. While we acknowledge that this is not the only possible model, the need to engage in a methodological dialogue is clear. Without higher order research frameworks systematic research in CER will ultimately lack power and credibility. We encourage further dialogue on the nature of the CER research paradigm.

References

- Berglund, A., Daniels, M., and Pears, A. (2006) Qualitative research projects in computing education research: An overview, *Australian Computer Science Communications*, vol. 28, no. 5, 25-34.
- Byram, M., Nichols, A., and Stevens, D. (red) (2001): *Developing Intercultural Competence in Practice*, Multilingual Matters Ltd., Clevedon, UK.
- Cajander, Å. (2010) Usability – who cares? The introduction of user-centred systems design in organisations. *Acta Universitatis Upsaliensis. Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 740*, Uppsala.
- Carr, W. and Kemmis, S. (1983) *Becoming Critical: Knowing Through Action Research*. Deakin University press, Melbourne.
- Coppola, N., Hiltz, S., and Rotter, N. (2004) Building Trust in Virtual Teams, *IEEE Transactions on Professional Communication*, vol. 47, 95-104.
- Crotty, M. (1998) *The Foundations of Social Research*, Sage publications, London.
- Daniels, M. (2011) *Developing and Assessing Professional Competencies: a Pipe Dream? Experiences from an Open-Ended Group Project Learning Environment*, *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 808*, Uppsala, Sweden.
- Elden, M. and Chisholm, R. (1993) Emerging varieties of action research: Introduction to the special issue, *Human relations*, vol. 46, no. 2, 121-142.
- Jarvenpaa, S., Knoll, K., and Leidner, D. (1998) Is Anybody Out There? Antecedents of Trust in Global Virtual Teams, *Journal of Management Information Systems*, vol. 14, 29-64.
- Laxer, C., Daniels, M., Cajander, Å., and Wollowski, M. (2009) Evolution of an International Collaborative Student Project, *Australian Computer Science Communications*, vol. 31, no. 5, 111-118.
- Lewin, K. (1946) Action research and minority problems, *Journal of social issues*, vol. 2, no. 4, 34-46.
- Maynard, M. (1994) Methods, practice, and epistemology: The debate about feminism and research, in *Researching women’s lives from a feminist perspective*, eds. Maynard and Purvis, Taylor and Francis, London, 10-26.
- McKay, J. and Marshall, P. (2001) The dual imperatives of action research, *Information Technology and People*, vol. 14, 46-59.
- Panteli, N. and Duncan, E. (2004) Trust and temporary virtual teams: alternative explanations and dramaturgical relationships, *Information Technology and People*, vol. 17, 423-441.

- Pears, A., Daniels, M., and Berglund, A. (2002) Describing Computer Science Education Research: An Ac
- Pears, A. and Daniels, M. (2003) Structuring CSEd research studies: Connecting the pieces, ACM SIGCSE Bulletin, vol. 35, no. 3, 149-153.
- Pears, A. and Daniels, M. (2003) Structuring CSEd research studies: Connecting the pieces, ACM SIGCSE Bulletin, vol. 35, no. 3, 149-153.
- Piaget, J. (1970) Science of Education and the Psychology of the Child, Orion.
- Rasmussen, L. (2004) Action research – Scandinavian experiences, AI and society, vol. 18. no. 1, 21-43.
- Reason, P. (2006) Choice and quality in action research practice, Journal of management inquiry, vol. 15, no. 2, 187-203.
- Reason, P. and Bradbury, H. (2007) Handbook of action research, Sage, London.
- Westlander, G. (2006) Researcher roles in action research, in Action an interactive research – Beyond practice and theory, eds Nielsen and Svensson, Shaker, Maastricht, 45-62.

Illustration of Paradigm Pluralism in Computing Education Research

Neena Thota

School of Intelligent Systems and
Technology
University of Saint Joseph
Macau, S.A.R.
neenathota@usj.edu.mo

Anders Berglund

Uppsala Computing Education
Research Group, UpCERG
Department of Information Technology
Uppsala University
Uppsala, Sweden
Anders.Berglund@it.uu.se

Tony Clear

School of Computing and
Mathematical Sciences
Faculty of Design and Creative
Technologies
AUT University
Auckland, New Zealand
tony.clear@aut.ac.nz

Abstract

This paper argues for paradigm pluralism in computing education research. The value of mixing paradigms, and the choice of methodological eclecticism and mixed methods is explored using pragmatic knowledge claims. A research study, which focused on the design of an introductory object-oriented programming (OOP) course for undergraduate students, is introduced as an illustration of paradigm pluralism. The study demonstrates methodological eclecticism and use of mixed methods for data collection and analysis. Meaningful outcomes resulting from the choice of the research design are described. A framework that focuses on the research problem and research questions to guide research design is presented as the outcome of the study. Through the discussion and demonstration of paradigm pluralism, this paper contributes to increased awareness of theoretically anchored research in computer science.

Keywords: Paradigm, methodology, mixed methods.

1 Introduction

Methodological issues are becoming more important as the field of Computing Education Research (CER) matures. The move from single-method studies to multi-method studies in all disciplines in the social and behavioural sciences over the past decade (Teddlie and Tashakkori, 2010) calls for a reinterpretation of the procedures for selecting research approaches in computing education. The need to clarify the intent for inclusion of multiple methods of data collection and multiple forms of analysis, and the complexity of designing multi-method studies, calls for more explicit procedures focused on understanding the research problem and the philosophical foundation for the choice of methodology. A pragmatic viewpoint (Johnson and Onwuegbuzie, 2004, Creswell and Plano Clark, 2007), which seeks appropriateness of research methods or approaches to answering the research question, is a suitable foci for the integration of quantitative and qualitative research strands.

The suitability and feasibility of multi-paradigm and mixed methods studies in computing education research has been discussed in the literature. A multi-paradigm approach to computer science education can provide a panoptic view (Greening, 1996) leading to valuable insights into teaching and learning within the computing discipline. Multi-method research can increase rigor through triangulation within a single study or across a series of studies (Fincher and Petre, 2004). The practicality of using mixed methods in computing education to conduct research in stages to answer quantitative or qualitative questions is evident (Hazzan et al., 2006). Computing educators are urged to adopt a pragmatic approach employing mixed methods, with triangulation of data from different sources, student grades, and student and teacher perspectives (Clear, 2001). An analysis of research papers (Sheard et al., 2009), published in computing education conferences in the years 2005 to 2008, found that mixed methods approaches were favoured for studies that investigated programming ability, aptitude, or understanding, and for those that dealt with teaching, learning, assessment techniques, or tools for programming.

Some examples of mixed research approaches and methods in CER studies can be found. Berglund (2005), who interviewed students in an international distributed computer course, used phenomenographic research approach to analyze the data and activity theory to synthesize the results. Meisalo et al. (2003) integrated qualitative analysis of interview data with statistical analysis of questionnaire data and logs of action to evaluate the study process in virtual programming courses. Kinnunen and Malmi (2005) coded observations, interviews, questionnaires, and course results into categories and sequence of numbers to analyze the interactions in problem-based learning. Soh et al. (2007) evaluated a framework for improving programming placements through examinations in a pre-test/post-test research design, laboratory assignments, and questionnaires to assess students' self-efficacy and motivation. However, the existing CER literature lacks a knowledge base that examines worldview stances and mixed method design considerations, and that provides an example of a carefully considered study which evaluates the methodological choices with an emphasis on standards. The interaction of paradigms, methodology, and methods has not been explored adequately in CER. This paper redresses the paucity of such a knowledge base by developing and demonstrating a framework for the design of multi-method studies.

The main contributions of this paper are: (a) the demonstration of a study that encompasses paradigm pluralism, methodological eclecticism, and mixed methods, increases awareness of how such studies can be conducted, and illustrates the kind of educational outcomes that such studies can be expected to generate; and (b) the framework contributes to advances in the discipline by being grounded in established educational practices and theory, and by providing a structured overview of the inter-disciplinary components of research necessary to address complex research situations.

The intent of this paper is to present the theoretical and methodological aspects of the design of the exemplar study, rather than discuss the findings in detail. The remainder of this paper is organised into 6 sections. Section 2 deals with the arguments for the adoption of a multi-paradigm, multi-method approach. The design considerations for a research framework are described in section 3. In section 4, an example of a research study is given as a demonstration of the design. Section 5 discusses the implications of using the design framework and the paper concludes with section 6.

2 Building the case

Paradigms or worldviews denote a set of beliefs about how we view the world and conduct research (Guba, 1990). Within worldviews, ontological assumptions give rise to epistemological perspectives, which guide methodological considerations and the determination of the choice of instrumentation, data collection methods, and data analysis techniques (Hitchcock and Hughes, 1989).

The emergence of different worldviews has led to the expansion of the paradigms of positivism and constructivism to now include critical theory, postpositivism, participatory research (Guba and Lincoln, 2005, Lincoln et al., 2011), pragmatism (Johnson and Onwuegbuzie, 2004, Creswell and Plano Clark, 2007), and transformative paradigm (Mertens, 2007). Paradigmatic strands of research can come together and generate the potential for multiple interpretive practices for the researcher who works between competing paradigms (Guba and Lincoln, 2005). Morgan's (2007) stance on paradigms as shared beliefs in a research community is echoed by Denzin's (2010, p. 420) call for a "new paradigm dialog" that transcends paradigms, methodologies, and epistemologies, and honours cooperation and collaboration among the community of scholars. *Paradigm pluralism* (Teddlie and Tashakkori, 2010) thus denotes the adoption of a variety of paradigms as the philosophical foundation for a study.

The *pragmatic* worldview (Morgan, 2007) is a deliberate choice for practitioners who practice a pluralistic orientation towards paradigms focused on the primary importance of the research question and multi-method data collection and analysis. Pragmatism, as a research paradigm, accepts multiple realities and orients itself toward solving practical problems (Creswell and Plano Clark, 2007). The tenets of pragmatism include the adoption of a value-oriented approach to research (Johnson and Onwuegbuzie, 2004). The pragmatic stance offers flexibility in addressing a range of research questions that arise, promotes collaboration among

researchers regardless of philosophical orientation, and enables the combination of empirical precision with descriptive precision (Onwuegbuzie and Leech, 2005).

A distinction can be made between *methodology*, which connotes a broad inquiry logic or general approach to an inquiry, and *methods*, which are specific techniques for design, sampling, data collection, data analysis, and interpretation of findings (Crotty, 1998). *Methodological eclecticism* (Johnson and Onwuegbuzie, 2004, Teddlie and Tashakkori, 2010) is the pragmatic selection and integration of qualitative and quantitative techniques to investigate a research problem. Methodological eclecticism stems from the choice of an ontology of multiple realities that repudiates the *incompatibility thesis* (Howe, 1988) which posits qualitative and quantitative research paradigms are mutually exclusive. Methodological eclecticism is a key feature of *mixed methods*, a practice of combining quantitative and qualitative research techniques, methods, approaches, concepts or language into a single study (Johnson and Onwuegbuzie, 2004).

The mixed methods approach embraces multiple philosophical paradigms and multiple ways of making sense of the world (Greene, 2008). Mixed methods provide quantitative and qualitative research strengths and enable a researcher to answer a broader and more complete range of research questions by drawing conclusions and inferences from convergent and divergent results (Teddlie and Tashakkori, 2010). Mixed methods can be exploratory, explanatory, confirmatory, action, transformative, or critical (Christ, 2009). Combining different research methods from different existing paradigms by using a critical pluralistic position enriches and adds to the reliability of results in multi-phase research studies (Mingers, 2001). Within the action research methodology, mixed methods allow researchers and participants to use a multiplicity of data collection instruments to accumulate evidence from multiple accounts (Cohen et al., 2007). Action research itself can be viewed as a form of mixed methods where the theoretical lens of critical realism can be applied to multiple forms of data (Christ, 2010).

3 Framework for a research study

Research models or frameworks have been suggested for the design of mixed methods studies. Collins et al. (2006) outlined 13 steps in three stages: (a) research formulation (determining the research goal, objectives, rationale, purpose, and research questions); (b) research planning (selecting the sampling and study design); and (c) research implementation (data collection, analysis, validation, and interpretation). These stages are followed by research dissemination and possible reformulation of the research question. Collins and O'Cathain (2009) later refined the stages to 10 steps. Many of the steps are considered sequential. However, research studies that include data collection from qualitative as well as quantitative methods are generally iterative as the phenomenon undergoes deeper levels of understanding when findings and inferences get synergistically integrated (Maxwell and Loomis, 2003, Teddlie and Tashakkori, 2010). The framework that this paper suggests, for a research study that spans paradigms and

methodologies, was the outcome of a cyclic research process that was integrative of the research steps from the research formulation, planning, and implementation stages.

Figure 1 depicts the research framework based on an explicit consideration of the research questions as a pragmatic guide to define the philosophical foundation and the development of the research design. The centrality of the research purpose, the underlying philosophical assumptions, and the research procedures constitute the elements of the study. The figure shows the interaction of the elements that can help researchers not only to clarify their conceptual foundations, but also to document their design choices.

In figure 1, the primacy of the researcher's theoretical, personal and/or professional goals in determining a problem and formulating researching questions is emphasized. The philosophical assumptions include the worldviews held by the researcher, the methodological choices, and the research validity and credibility criteria that stem from the conceptual orientations. The research purposes and the underlying philosophical assumptions determine the nature of the design typology, and the selection of the data collection and analysis methods that a researcher applies. The synthesis of the conclusions and inferences from the study stems from the interactive nature of the various research study elements.

4 Example of research study

The research study (Thota, 2011) discussed in this paper tracked the iterative design, implementation, and evaluation of an introductory object-oriented programming (OOP) course using the java programming language. In addition, to emphasizing constructive alignment of outcomes and assessments, use of variation theory, and the utilization of learning technologies in the first iteration, the course design focused on balancing theoretical with experiential understanding, on building connections with students, and on the deliberate inclusion of student perspectives in the course design in the second iteration.

The OOP course was taught in two semesters (2008 to 2009) to first year programming students in the University of Saint Joseph, Macau, which is affiliated to the Catholic University of Portugal. The students were majors in Information Systems, Business Technology Management, Business Administration, and Design. Twenty six students in the first iteration and 72 students in the second iteration participated in the research study.

Figure 2 shows the research framework for the study, laid out in block diagram format to aid readability. The research purposes, the philosophical assumptions and the design procedures in the figure are described in the following sub-sections.

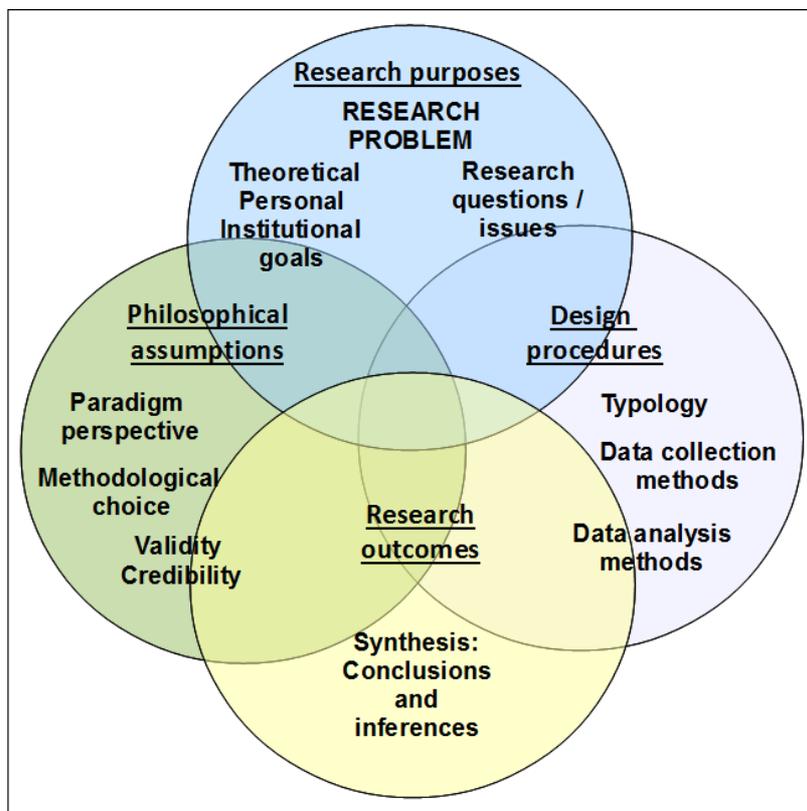


Figure 1: Framework for research study with mixed paradigms, methodologies, and methods.

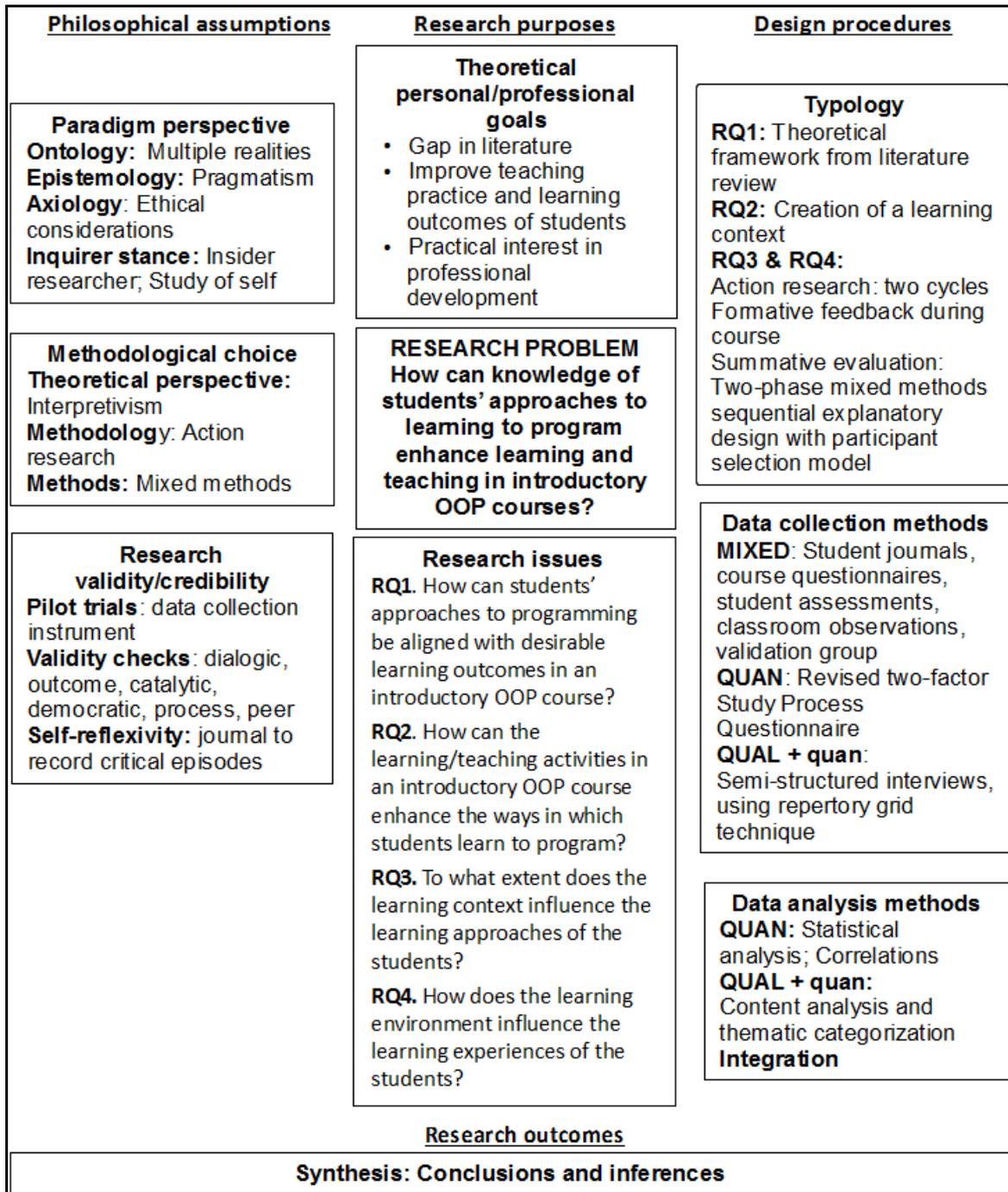


Figure 2: Framework with research purposes, philosophical assumptions, and design procedure

4.1 Research purposes

Theoretical, personal, and pragmatic orientations led to the identification of the research problem that was tackled in this study:

How can knowledge of students' approaches to learning to program enhance learning and teaching in introductory OOP courses?

A review of the literature had revealed that learning programming is a perennial problem that continues to be discussed (Carbone et al., 2009, Robins, 2010). A number of studies did exist on factors that influence learning in programming (Chamillard, 2006, Rountree et al., 2004, Wiedenbeck, 2005, Bergin and Reilly, 2006) as well as attempts to improve learning through changes in teaching

strategies (Soh et al., 2007, Caspersen and Bennedsen, 2007, Gries, 2008). However, course designs have failed to incorporate phenomenographic research findings of students' approaches and conceptions of programming (Marton and Booth, 1997, Booth, 1992), specific attention to the critical aspects of learning programming (Bruce et al., 2004, Eckerdal and Berglund, 2005), and the influence of the learning and teaching context on students' learning approaches (Biggs, 1987, Ramsden, 2005).

A coherent course design for object oriented programming (OOP), that was founded in an awareness of how students learn to program and which incorporated the technological demands and the needs of novice

programmers, did not exist. It was evident that there was a requirement for developing a teaching environment conducive to the adoption of deep learning approaches leading to successful learning outcomes in OOP.

The background of the field established the context and purpose for the research. However, the researcher's personal situation provided the motivation for the inquiry. The contradiction between the researcher's passion for the subject and the varied experiences of the students strengthened the resolve to improve the teaching practice and the learning outcomes of the students.

Many institutional factors also influenced the initiation of the research study. There was a need for a course design that integrated outcomes, assessments, teaching, and learning activities to motivate students from mixed majors taking the introductory programming course. The advances in information technology necessitated the integration of OOP software, visualization, and animation tools with the technological infrastructure of the university. There was also the expectation that the students, who hailed from multi-cultural backgrounds, should be trained to participate in distributed and collaborative programming projects. Thus, professional and institutional considerations provided pragmatic impetus for the research. The research issues, which were identified for investigation, can be seen in figure 2 and are discussed further in the next section.

4.2 Philosophical assumptions

Paradigm perspective (Crotty, 1998) can be explicated in terms of the researcher's stance on the nature of reality (ontology), the nature of knowledge (epistemology), and ethics and values (axiology). The beliefs and basic elements that underpinned this study were pragmatically driven. The deliberate choice of ontology of multiple realities led to the adoption of a pragmatic approach to a research design that favoured methodological appropriateness (Patton, 1990). Action research, as a methodological choice, was considered suitable for producing both personal action and theoretical research as intended outcomes (Dick, 1997), and served as the interface between the underlying theory and the choice of mixed methods. The adoption of an interpretivist stance emphasized that realities are multiple, constructed and holistic, that knowledge was jointly constructed by the participants and the researcher, and that the inquiry was value laden (Lincoln and Guba, 1985).

The pragmatic link to the research questions is described below.

RQ1. How can students' approaches to programming be aligned with desirable learning outcomes in an introductory OOP course?

A theoretical framework derived from the literature review (Thota and Whitfield, 2010) was devised for:

- Constructive alignment of intended learning outcomes with assessment tasks;
- Design of learning and teaching activities to encourage students to use deep learning approaches to achieve the learning outcomes.

RQ2. How can the learning/teaching activities in an introductory OOP course enhance the ways in which students learn to program?

The theoretical framework, derived from the literature review, was further extended for:

- Creation of a learning context to enable students to experience a variety of educationally critical ways of learning to program;
- Creation of a learning context with multiple media to enhance the learning experiences.

The action research project, with two cycles, was planned for the implementation of the OOP course and to investigate the remaining research questions:

RQ3. To what extent does the learning context influence the learning approaches of the students?

RQ4. How does the learning environment influence the learning experiences of the students?

The methodological decision to pursue action research was grounded in the notion of a self-reflective practitioner intent on rigorous research (McNiff and Whitehead, 2002) through cycles of planning, action, observation, reflection, and evaluation. The hallmark of the study was the adoption of a pragmatic-constructivist approach to connect theory and data, the focus on the intersubjectivity of the relationships in the research process, and the acceptance of transferability of inference from quantitative and qualitative data (Morgan, 2007). The researcher's position as an insider (Anderson and Herr, 2005) established that there was no separation of the study of practice and self, from the study of the outcomes of the actions that were initiated (Bullough and Pinnegar, 2001).

In this study, all claims to improvement were based solely on the researcher's professional judgment. Formative, summative, and illuminative evaluations (Jacobs, 2000), inclusive of reflection during and after the practice (Schön, 1983), were undertaken to assess the outcomes of the action research project. The study itself was evaluated using criteria uniquely suited to the purposes and procedures of practitioner research, rather than by criteria established within other paradigms. A set of validity criteria (Anderson and Herr, 1999) that are linked to the goals of insider action research (dialogic, outcome, catalytic, democratic, and process) were applied as summative evaluation of the research study. Implicit in these standards of judgment are the processes of personal, empathetic, social, institutional, and ethical validity that can be found in the works of theorists such as Lather (1986), Winter (1996), and McNiff and Whitehead (2002).

Issues of rigour and reflexivity were addressed through pilot trials that were undertaken to assess the functionality of the mixed methods data-gathering techniques, and by writing a reflective journal on the critical episodes (McNiff et al., 1996) in the research process. Informed consent was obtained from the participants, and the nature of the research was disseminated to all participants (Cohen et al., 2007).

4.3 Design procedures

Within the action research project, two cycles of planning and acting led to the implementation of the OOP course. To provide authentic descriptions of the action (McNiff et al., 1996), student and teacher artefacts were incorporated as data sources for the study. Formative feedback and summative evaluation provided ways to gather data for research questions 3 and 4. In each cycle, formative feedback was obtained from:

- The reflective journals that students wrote about their course experience.
- Data from course questionnaires that the students answered during the teaching period. (A range of influences on learning outcomes was investigated including prior knowledge, perceptions of learning to program, motivation and self-efficacy levels, beliefs about collaborative work, and views about technologies.)
- The grades obtained by students on the programming quizzes, exam, assignments, and project.
- Observations of classroom interactions.
- Feedback from validation groups: critical friend, student tutor, and colleagues at the university.

The students' programming experiences (from journals) were interpreted and categorized through thematic analysis (Ezzy, 2002, Patton, 1990). Within each cycle, the two-phase mixed methods sequential explanatory design with participant selection model (Creswell and Plano Clark, 2007) was utilized to gather data for the summative evaluation. The linkages between the research questions and the data collection and analysis methods are outlined next.

RQ3. To what extent does the learning context influence the learning approaches of the students?

In the mixed methods quantitative phase, the data collection instrument and analysis procedures were:

- The two-factor Revised Study Process Questionnaire (R-SPQ-2F), which is grounded in student learning theories, to identify students' learning approaches (Biggs et al., 2001).
- Computation of deep/surface learning approach scores; correlation of approach scores with course grades; correlations of course grades with exam marks.
- Identification of a cross-section of students purposefully selected for a follow-up, in-depth study of their perceptions of the learning environment. (Students who obtained the highest and lowest scores on the correlated measures were identified, and these students were invited for interviews in the next primarily qualitative phase of the study.)

RQ4. How does the learning environment influence the learning experiences of the students?

In the mixed methods primarily qualitative phase, the data collection instrument and analysis procedures were:

- Semi-structured interviews, using the repertory grid technique (Kelly, 1955), to elicit views about the phenomenon under investigation. (The technique is grounded in personal construct theory and yields qualitative and numeric data.)

- Collection of the descriptive constructs and numeric ratings from the repertory grids.
- Data transformation using Honey's (1979) content analysis technique.
- Thematic categorization of the qualitized data, inductively analyzed to identify themes (Ezzy, 2002, Patton, 1990).

The mixed methods design was characterized by the use of quantitative participant characteristics to guide purposeful sampling for the primarily qualitative second phase. The purpose of the two phases in the research study was to investigate the learning approaches of the students, and then to understand the ways in which students with different learning approaches experienced the OOP course. Figure 3 shows the research design with mixed methods embedded in the action research study.

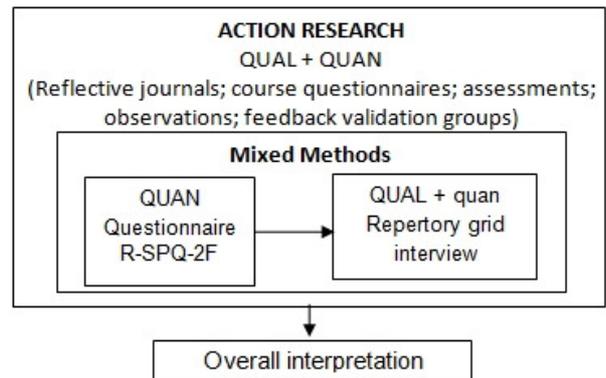


Figure 3: Action research study with mixed methods.

Note. QUAL stands for qualitative; QUAN/quan stands for quantitative. Capital letters denote high priority or weight; lower case letters denote lower priority or weight; → stands for sequential process. Adapted from Morse (2003).

Since the goal of this study was not to generalize to other contexts, but to obtain insights into the programming phenomenon, participants who had experienced the central phenomenon of introductory programming were selected purposefully by utilizing a homogenous sampling scheme (Patton, 1990). The population thus comprised first year undergraduate students representing homogenous characteristics i.e. studying in the introductory Programming Concepts course. In cycle one, 21 of the 26 students enrolled in the course answered the questionnaire. Fourteen students were interviewed using the repertory grid technique. In cycle two, 72 of the 85 students enrolled in the course answered the questionnaire, and 15 students were interviewed. The theoretical lens through which the analyses, research practices, and conclusions were presented was mainly interpretivist (Crotty, 1998). The opportunity to synthesize the results from the action research cycles led to meta-inferences from the qualitative and quantitative data that was gathered.

5 Discussion

In this study, the adoption of a multi-paradigm approach, grounded in epistemology and pedagogy, led to workable solutions that were related to the theoretical, personal, and professional goals of the researcher. The emphasis on an approach driven by research questions determined the

specific methods of data collection that informed the problem under study. The quality criteria used to evaluate the outcomes of the research showed how effectively the inferences answered the research purposes.

The development of an explicit research model while investigating a research problem has the potential to enhance the relevance, worth, and applicability of the research (Pears and Daniels, 2003). The framework that was developed in this study was in itself an outcome of the research process and led to valuable results. The framework can be refined and adapted for different settings with different research questions.

The choice of methodological eclecticism in this research study enabled insights that would not have been possible with a dogmatic stance. Action research afforded an appropriate methodology in a study aimed at iterative improvements in teaching and learning introductory programming. Learning and contextual issues were identified from the feedback that was received during the course, and proved valuable for understanding and complementing the data which was gathered as summative evaluation at the end of the course. The findings from the first action research cycle served to inform the course redesign in the second cycle. The findings from the second cycle acted as beacons for future development. Doing and writing about the action research project produced knowledge that was grounded in the lived experience of the situation, was co-contributed by the student participants and researcher, and validated through peer and public scrutiny (McNiff and Whitehead, 2009).

In this study, the quantitative data about the approaches of the students relating to the programming course, and the subjective interpretation of their experiences (qualitatively determined, and statistically and qualitatively interpreted through the repertory grid data analysis) made the inferences from the study much stronger. The utilization of mixed methods allowed enrichment and triangulation with self-reported data that showed how students approached and experienced the programming course. The collection of the qualitative data in the form of multiple perspectives and divergent views allowed an understanding of the varied ways of experiencing the phenomena under discussion. The use of a standard well validated questionnaire to identify learning approaches, and the use of the repertory grid interviews to elicit personal constructs about the learning experience served the purposes of complementarity and expansion (Greene et al., 1989). Complementarity led to elaboration and enhancement from the methods to increase the interpretability and meaningfulness of the questionnaire results and the personal constructs of the students, while the breadth and scope of the research was expanded by using different methods for different research issues.

The rationale and purposes (Collins et al., 2006), for using mixed methods in this study led to (a) participant enrichment (students were selected with clearly identifiable surface and deep approaches for interviews about their learning experiences); (b) treatment integrity (fidelity with the underlying theory and principles of constructive alignment and phenomenographic pedagogy

guiding the course design); and (c) significance (thick, rich data from the qualitative and quantitative data collection methods).

The emergence of both convergent and divergent results from the data analysis provided greater insights into the phenomenon of teaching and learning introductory programming and opened up previously unexplored aspects of the research. The findings from the observations and reflective journals suggested that students underwent transformations in their thinking about programming, which was not obvious from the statistical data collected from the questionnaire. The journals that the students submitted during the course also contained rich descriptions that shed light on the tacit understandings of novice programmers. These conceptions informed the course developer to design meaningful learning and teaching activities to encourage reflective thinking about programming.

The numeric scores from the questionnaires revealed that students employed a range of learning approaches depending on the contextual influences they perceived as assisting learning to program. The findings open the way for further investigations about the learning approaches of novice programmers in cross-cultural situations.

The qualitative and quantitative findings from the repertory grid interviews revealed that the students found the learning process (reflection and experiencing), learning content (information, coding, assessment), and learning support (scaffolding and collaboration) helpful for programming. The students' constructs of their course experience served to improve the course developer's understanding of the contextual influences on students' learning. This understanding was then rechanneled to make improvements to the learning environment.

Using mixed methods in action research is challenging, as both the quantitative and qualitative strands bring their own unique challenges to the study (Collins et al., 2007). With respect to the quantitative instrument, the sample sizes in this study were too small to detect statistically significant differences or relationships. The *crisis of representation* (Denzin and Lincoln, 2005) in the qualitative strand was the challenge to capture the lived experience of the participants in textual format. Therefore, a rigorous procedure for developing the thematic categories for construct analysis was established with three independent coders achieving acceptable levels of inter-rater reliability for agreement of construct categories.

Researchers who blend methodologies when they study their own practice bear the onus of establishing scholarly integrity as writers and methodologists (Bullough and Pinnegar, 2001). In this study, evidence of scholarly writing was presented by recording participants' thinking and feelings in an authentic manner, and by selecting, framing, and evaluating the outcomes of the study within a written document (Thota, 2011), that provided the structure and coherence for argumentation. The development of some necessary skills for repertory grid practitioners (Fransella, 2005), such as credulous listening, reflexivity, and construct interpretation, became part of the learning experience for the researcher.

6 Conclusion

In this paper, an argument was put forth for paradigm pluralism and methodological eclecticism in computing education research. Pragmatism was advocated for its practical relevance to mixing paradigms and methods. A framework with a focus on the research problem and research questions to guide methodological choices was presented. The detailed description of a mixed methods research design, along with a discussion of issues, was provided to illustrate the framework.

Emerging trends in integrated research methodology necessitate that computing education researchers are conversant with contemporary orientations in mixing paradigms and methods. Through the discussion and demonstration of a multi-paradigm approach and mixed methods data collection design embedded in practitioner-led action research, this paper contributes to furthering a scholarly enquiry and methodological awareness among computing educators. The design of this research study merits consideration by educators who wish to understand how mixing paradigms, the adoption of methodological eclecticism, and the use of mixed methods can be utilised to investigate research issues in computing education.

7 References

- Anderson, G. L. & Herr, K. (1999): The new paradigm wars: Is there room for rigorous practitioner knowledge in schools and universities? *Educational Researcher*, **28**(5): 12-40.
- Anderson, G. L. & Herr, K. (2005): *The action research dissertation: A guide for students and faculty*. Thousand Oaks, CA, Sage.
- Bergin, S. & Reilly, R. (2006): Predicting introductory programming performance: A multi-institutional multivariate study. *Computer Science Education*, **16**(4): 303-323.
- Berglund, A. (2005): *Learning computer systems in a distributed project course: The what, why, how and where*. Uppsala, Sweden, Acta Universitatis Upsaliensis.
- Biggs, J. B. (1987): *Student approaches to learning and studying*. Hawthorn, Victoria, Australian Council for Educational Research.
- Biggs, J. B., Kember, D. & Leung, D. (2001): The revised two-factor Study Process Questionnaire: R-SPQ-2F. *British Journal of Educational Psychology*, **71**(1): 133-149.
- Booth, S. (1992): *Learning to program: A phenomenographic perspective*. Göteborg, Sweden, Acta Universitatis Gothoburgensis.
- Bruce, C., McMahon, C., Buckingham, L., Hynd, J., Roggenkamp, M. & Stoodley, I. (2004): Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education*, **3**: 143-160.
- Bullough, R. V., Jr. & Pinnegar, S. (2001): Guidelines for quality in autobiographical forms of self-study research. *Educational Researcher*, **30**(3): 13-21.
- Carbone, A., Hurst, J., Mitchell, I. & Gunstone, D. (2009): An exploration of internal factors influencing student learning of programming. In *Proc. 11th Australasian Computing Education Conference (ACE 2009)*. 25-34. Hamilton, M. & Clear, T. (eds). Darlinghurst, Australia, ACS.
- Caspersen, M., E & Bennedsen, J. (2007): Instructional design of a programming course: A learning theoretic approach. *Proc. Third International Computing Education Research Workshop (ICER 2007)*, Atlanta, GA, 111-122, ACM.
- Chamillard, A. T. (2006): Using student performance predictions in a computer science curriculum. *Proc. 11th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2006)*, Bologna, Italy, 260-264, ACM.
- Christ, T. W. (2009): Designing, teaching, and evaluating two complementary mixed methods research courses. *Journal of Mixed Methods Research*, **3**(4): 292-325.
- Christ, T. W. (2010): Teaching mixed methods and action research: Pedagogical, practical, and evaluative considerations In *Handbook of Mixed Methods in Social & Behavioral Research*. 643-676. Tashakkori, A. & Teddlie, C. (eds). 2nd ed. Thousand Oaks, CA, Sage.
- Clear, T. (2001): Research paradigms and the nature of meaning and truth. *ACM SIGCSE Bulletin*, **33**(2): 9-10.
- Cohen, L., Manion, L. & Morrison, K. (2007): *Research methods in education*. New York, NY, Routledge.
- Collins, K. M. T. & O'Cathain, A. (2009): Introduction: Ten points about mixed methods research to be considered by the novice researcher. *International Journal of Multiple Research Approaches*, **3**(1): 2-7.
- Collins, K. M. T., Onwuegbuzie, A. J. & Jiao, Q. G. (2007): A mixed methods investigation of mixed methods sampling designs in social and health science research. *Journal of Mixed Methods Research*, **1**(3): 267-294.
- Collins, K. M. T., Onwuegbuzie, A. J. & Sutton, I. L. (2006): A model incorporating the rationale and purpose for conducting mixed-methods research in special education and beyond. *Learning Disabilities: A Contemporary Journal*, **4**(1): 67-100.
- Creswell, J. W. & Plano Clark, V. L. (2007): *Designing and conducting mixed methods research*. Thousand Oaks, CA, Sage.
- Crotty, M. (1998): *The foundations of social research: Meaning and perspective in the research process*. Thousand Oaks, CA, Sage.
- Denzin, N. & Lincoln, Y. (2005): Introduction: The discipline and practice of qualitative research. In *Handbook of Qualitative Research*. 1-32. Denzin, N. K. & Lincoln, Y. S. (eds). 3rd ed. Thousand Oaks, CA, Sage.
- Denzin, N. K. (2010): Moments, mixed methods, and paradigm dialogues. *Qualitative Inquiry*, **16**(6): 419-427.
- Dick, B. (1997). *Approaching an action research thesis: An overview*. http://www.uq.net.au/action_research/arp/phd.html. Accessed 20 Aug 2011.

- Eckerdal, A. & Berglund, A. (2005): What does it take to learn 'programming thinking'? *Proc. First International Computing Education Research Workshop (ICER 2005)*, Seattle, WA, USA, 135-142, ACM.
- Ezzy, D. (2002): *Qualitative analysis: Practice and innovation*. London, UK, Routledge.
- Fincher, S. & Petre, M. (Eds). (2004): *Computer science education research*. London, UK, RoutledgeFalmer.
- Fransella, F. (2005): Some skills and tools for personal construct users. In *The Essential Practitioner's Handbook of Personal Construct Psychology*. 41-56. Fransella, F. (ed). West Sussex, UK, Wiley.
- Greene, J. C. (2008): Is mixed methods social inquiry a distinctive methodology? *Journal of Mixed Methods Research*, **2**(1): 7-22.
- Greene, J. C., Caracelli, V. J. & Graham, W. F. (1989): Toward a conceptual framework for mixed-method evaluation designs. *Educational Evaluation and Policy Analysis*, **11**(3): 255-274.
- Greening, T. (1996): Paradigms for educational research in computer science. *Proc. Second Australasian Conference on Computer Science Education (ACSE)*, University of Melbourne, Australia, 47-51, ACM Press.
- Gries, D. (2008): A principled approach to teaching OO first. *Proc. 39th SIGCSE Technical Symposium on Computer Science Education*, Portland, OR, USA, 31-35, ACM.
- Guba, E. (1990): The alternative paradigm dialog. In *The Paradigm Dialog*. 17-27. Guba., E. G. (ed). Newbury Park, CA, Sage.
- Guba, E. & Lincoln, Y. (2005): Paradigmatic controversies, contradictions, and emerging confluences. In *Handbook of Qualitative Research*. 191-215. Denzin, N. K. & Lincoln, Y. S. (eds). 3rd ed. Newbury Park, CA, Sage.
- Hazzan, O., Dubinsky, Y., Eidelman, L., Sakhnini, V. & Teif, M. (2006): Qualitative research in computer science education. *ACM SIGCSE Bulletin*, **38**(1): 408-412.
- Hitchcock, G. & Hughes, D. (1989): *Research and the teacher*. Routledge Kegan & Paul.
- Honey, P. (1979): The repertory grid in action: How to use it to conduct an attitude survey. *Industrial and Commercial Training*, **11**(11): 452-459.
- Howe, K. (1988): Against the quantitative-qualitative incompatibility thesis or dogmas die hard. *Educational Researcher*, **17**(8): 10.
- Jacobs, C. (2000): The evaluation of educational innovation. *Evaluation*, **6**(3): 261-280.
- Johnson, R. B. & Onwuegbuzie, A. J. (2004): Mixed methods research: A research paradigm whose time has come. *Educational Researcher*, **33**(7): 14-26.
- Kelly, G. A. (1955): *The psychology of personal constructs*. New York, NY, Norton.
- Kinnunen, P. & Malmi, L. (2005): Problems in problem-based learning- Experiences, analysis and lessons learned on an introductory programming course. *Informatics in Education*, **4**(2): 193-212.
- Lather, P. (1986): Issues of validity in openly ideological research: Between a rock and a soft place. *Interchange*, **17**(4): 63-84.
- Lincoln, Y. S. & Guba, E. G. (1985): *Naturalistic inquiry*. Newbury Park, CA, Sage.
- Lincoln, Y. S., Lynham, S. A. & Guba, E. G. (2011): Paradigmatic controversies, contradictions, and emerging confluences, revisited. In *Handbook of Qualitative Research*. 97-128. Denzin, N. K. & Lincoln, Y. S. (eds). 4th ed. Thousand Oaks, CA, Sage.
- Marton, F. & Booth, S. (1997): *Learning and awareness*. Mahwah, NJ, Laurence Erlbaum Associates.
- Maxwell, J., A. & Loomis, D., M. (2003): Mixed methods design: An alternative approach. In *Handbook of Mixed Methods in Social and Behavioral research*. 241-272. Tashakkori, A. & Teddlie, C. (eds). Thousand Oaks, CA.
- McNiff, J., Lomax, P. & Whitehead, J. (1996): *You and your action research project*. London, UK, RoutledgeFalmer.
- McNiff, J. & Whitehead, J. (2002): *Action research: Principles and practice*. London, UK, RoutledgeFalmer.
- McNiff, J. & Whitehead, J. (2009): *Doing and writing action research*. London, UK, Sage.
- Meisalo, V., Sutinen, E. & Torvinen, S. (2003): Choosing appropriate methods for evaluating and improving the learning process in distance programming courses. *Proc. 33rd ASEE/IEEE Frontiers in Education Conference*, Boulder, Colorado, 11-16.
- Mertens, D., M. (2007): Transformative paradigm: Mixed methods and social justice. *Journal of Mixed Methods Research*, **1**(3): 212-225.
- Mingers, J. (2001): Combining IS research methods: Towards a pluralist methodology. *Information Systems Research*, **12**(3): 240-259.
- Morgan, D., L. (2007): Paradigms lost and pragmatism regained: Methodological implications of combining qualitative and quantitative methods. *Journal of Mixed Methods Research*, **1**(1): 48-76.
- Morse, J. M. (2003): Principles of mixed methods and multimethod research design. In *Handbook of Mixed Methods in Social & Behavioral Research*. 189-208. Tashakkori, A. & Teddlie, C. (eds). Thousand Oaks, CA, Sage.
- Onwuegbuzie, A. J. & Leech, N. L. (2005): On becoming a pragmatic researcher: The importance of combining quantitative and qualitative research methodologies. *International Journal of Social Research Methodology*, **8**(5): 375-387.
- Patton, M. Q. (1990): *Qualitative evaluation and research methods*. Newbury Park, CA, Sage.
- Pears, A. & Daniels, M. (2003): Structuring CSEd research studies: Connecting the pieces. *Proc. Eighth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2003)*, Thessaloniki, Greece, 149-153, ACM Press.
- Ramsden, P. (2005): The context of learning in academic departments. In *The Experience of Learning:*

- Implications for Teaching and Studying in Higher Education* 198-216. Marton, F., Hounsell, D. & Entwistle, N. (eds). 3rd Internet ed. Edinburgh, University of Edinburgh, Centre for Teaching, Learning and Assessment.
- Robins, A. (2010): Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education*, **20**(1): 37-71.
- Rountree, N., Rountree, J., Robins, A. & Hannah, R. (2004): Interacting factors that predict success and failure in a CS1 course. *ACM SIGCSE Bulletin*, **36**(4): 101-104.
- Schön, D. A. (1983): *The reflective practitioner: How professionals think in action*. New York, NY, Basic Books.
- Sheard, J., Simon, Hamilton, M. & Lönnberg, J. (2009): Analysis of research into the teaching and learning of programming. *Proc. Fifth International Workshop on Computing Education Research (ICER 2009)*, Berkeley, CA, USA, 93-104, ACM.
- Soh, L., Samal, A. & Nugent, G. (2007): An integrated framework for improved computer science education: Strategies, implementations, and results. *Computer Science Education*, **17**(1): 59-83.
- Teddlie, C. & Tashakkori, A. (2010): Overview of contemporary issues in mixed methods research. In *Handbook of Mixed Methods in Social & Behavioral Research*. 1-41. Tashakkori, A. & Teddlie, C. (eds). 2nd ed. Thousand Oaks, CA, Sage.
- Thota, N. & Whitfield, R. (2010): Holistic approach to learning and teaching introductory object-oriented programming. *Computer Science Education*, **20**(2): 103-127.
- Thota, N. (2011): Developing a holistic approach to learning and teaching introductory object-oriented programming. Ph.D. thesis. University of Saint Joseph, Macau.
- Wiedenbeck, S. (2005): Factors affecting the success of non-majors in learning to program. *Proc. First International Computing Education Research Workshop (ICER 2005)*, Seattle, WA, USA, 13-24, ACM Press.
- Winter, R. (1996): Some principles and procedures for the conduct of action research. In *New Directions in Action Research*. 13-27. Zuber-Skerritt, O. (ed). Washington, D.C., Falmer Press.

Switch's CAM Table Poisoning Attack: Hands-on Lab Exercises for Network Security Education

Zouheir Trabelsi

Faculty of Information Technology
UAE University
Al-Ain, UAE

trabelsi@uaeu.ac.ae

Abstract

Teaching offensive techniques is a necessary component of a computer security education and yields better security professionals than teaching defensive techniques alone. In this paper, we describe a case study of the implementation of comprehensive hands-on lab exercises that are essential to security education. The first hands-on lab exercise is about how to perform a Denial of Service (DoS) attack based on the poisoning of the CAM tables (Content Access Memory) of Local Area Network (LAN) switches. The second exercise is about how to prevent CAM table poisoning attack. The hands-on labs confirmed further the ethical and legal concerns regarding the teaching of offensive techniques in the academic environment. In fact, the number of injected malicious traffic targeting the university switches' CAM tables, increased considerably each time the students experiment the DoS attack. That is why every course in IT security should be accompanied by a basic discussion of legal implications and ethics.

Keywords: Switch CAM table poisoning, DoS attack, Security port.

1 Introduction

Network security courses are often taught as concepts, at relatively abstract levels. A curriculum that covers the concepts of network security without giving suitable coverage to practical implementation deprives the student of the opportunity to experience the technologies and techniques required to ensure security. A hands-on approach to disseminating knowledge of network security will prepare the student for the complexities of conducting research and development in this field. Such an approach is rarely seen in most graduate and undergraduate courses. Even when the hands-on approach is advocated, by some, it is usually dominated by exercises using defensive techniques.

Recently, offensive techniques, originally developed by hackers, are gaining widespread approval and interest

(Yuan, and Zhong 2008, Bishop 1997, Frincke 2003, Hill, Carver, Humphries, and Pooch 2001, Mullins, Wolfe, Fry, Wynters, Calhoun, Montante, and Oblitey 2002). It is often criticized that offensive methods should not be taught to students as this only increases the population of "malicious hackers". Many educators in this field feel that hands-on courses that teach security attacks in detail are unethical, and create the potential for some to use the tools and techniques in an irresponsible manner (Harris 2004, Caltagirone, Ortman, Melton, Manz, King, and Oman 2006, Livermore 2007). The social implication is to restrict the injection of new hackers into society.

However, others claim that teaching offensive techniques yields better security professionals than those that are taught only defensive techniques (Mink and Freiling 2006, Arce and McGraw 2004, Arnett and Schmidt 2005, Dornseif, Holz, and Mink 2005, Vigna 2003, Yuan, Matthews, Wright, Xu, and Yu 2010, Livermore 2007). It is important to note that the corporate businesses employ experts that use offensive techniques for penetration testing, to ensure their security. The use of offensive techniques to provide secure environments for large corporate entities has created the new genre of hackers, the "ethical hacker"!

We believe that offensive techniques are central, to better understand security breaches and system failures. Teaching network attacks with hands-on experiments is a necessary component of education in network security. Moreover, we believe that security students need to experiment attack techniques to be able to implement appropriate and efficient security solutions. This approach to education will enable the student to provide confidentiality, integrity, and availability for computer systems, networks, resources, and data. One cannot perfectly design or build defenses for attacks that one has not truly experienced, first-hand. However, we agree that offensive techniques must not be taught as the primary focus of a course. Every course in IT security must be accompanied by discussion of legal implications and cover the ethical responsibilities of the student towards their community and society at large.

Network security lacks sufficient and contemporary textbooks and technical papers that describe in detail hands-on exercises that include both offensive and defensive implementation within an isolated network laboratory environment. To contribute to fill this void in security education, this paper proposes comprehensive hands-on lab exercises that are essential to security education. The first lab exercise is about how to perform

Copyright © 2012, Australian Computer Society, Inc. This paper appeared at the 14th Australasian Computing Education Conference (ACE 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 123. M. de Raadt and A. Carbone, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

a DoS attack based on the poisoning of the CAM table of a LAN switch. The second lab exercise is about how to prevent the poisoning of the switch's CAM table. The lab exercises allow students to better anatomize and elaborate the discussed offensive and defensive techniques. The lab exercises can be offered to students during security courses related to intrusion detection and prevention techniques, particularly to DoS attacks. It is designed to accompany and complement any existing trade or academic press text.

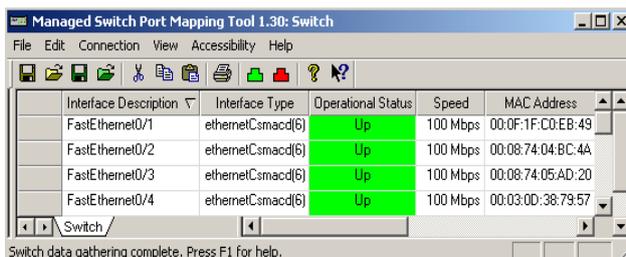
The paper is organized as follows: Section 2 includes a brief understanding of switch's CAM table, to form the base for subsequent sections. Section 3 discusses the first hands-on lab. Section 4 discusses the second hands-on lab. Section 5 discusses some ethical concerns related to teaching offensive techniques. Section 6 discusses the student satisfaction and the effect of offering the hands-on lab exercises on the student performance. Finally, Section 7 concludes the paper.

2 Background: Switch's CAM Table

To form the base for subsequent sections, this section includes a brief understanding of the switch's CAM table.

LAN's switches maintain a table called the CAM table, and maps individual MAC addresses on the network to the physical ports on the switch. This allows the switch to direct data out of the physical port where the recipient is located, as opposed to indiscriminately broadcasting the data out of all ports as a hub does. The advantage of this method is that data is bridged exclusively to the network segment containing the computer that the data is specifically destined for.

Figure 1 shows an example of entries in the CAM table of a switch. Four hosts are connected to the switch. For example, the first host (whose MAC address is 00:0F:1F:C0:EB:49) is connected to Port #1 (Interface: *FastEthernet0/1*) on the switch.



Interface Description	Interface Type	Operational Status	Speed	MAC Address
FastEthernet0/1	ethernetCsmacd(6)	Up	100 Mbps	00:0F:1F:C0:EB:49
FastEthernet0/2	ethernetCsmacd(6)	Up	100 Mbps	00:08:74:04:BC:4A
FastEthernet0/3	ethernetCsmacd(6)	Up	100 Mbps	00:08:74:05:AD:20
FastEthernet0/4	ethernetCsmacd(6)	Up	100 Mbps	00:03:0D:38:79:57

Figure 1: The entries of a CAM table

When the switch receives a packet from a host, it extracts first the destination MAC address from the header of the Ethernet frame. Using this MAC address, the switch gets the corresponding port number from the CAM table. Then, the packet is sent only to the host connected to that port. Therefore, even by setting a computer's network interface card (NIC) into the promiscuous mode, sniffing traffic in a switched LAN network is not possible. However, hackers use the Man-in-the-Middle (MiM) attack technique to intercept and sniff traffic in switched LAN network (SwitchSniffer 2011, Winarp 2011, and WinArpAttacker 2011).

3 Lab exercise: DoS attack based on CAM Table Poisoning

This hands-on lab exercise is about DoS attack using CAM table poisoning technique. The learning objective of this lab exercise is for students to learn how to poison the CAM table of a LAN's switch in order to perform DoS attack on target LAN's hosts.

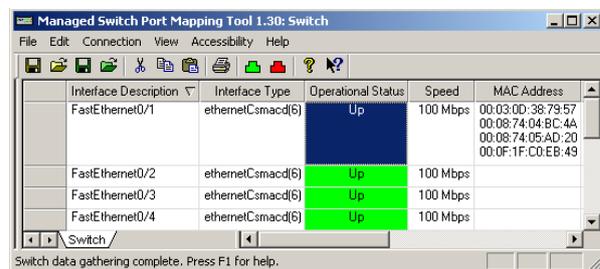
3.1 Attack Description

This attack intends to corrupt the entries in the switch's CAM table, so that the network traffic will be redirected. That is, a malicious host (connected to Port #a in a switch), sends a fake packet, with the source MAC address in the packet's Ethernet header set to the MAC address of a target host (connected to Port #b). The destination MAC address in the packet's Ethernet header can be any address. Once the switch receives the packet, it updates its CAM table. Therefore, the CAM table's entry for that target host's MAC address will be corrupted. Hence, the target host will be considered as a host connected to Port #a. Any packet sent to the target host (destination MAC address in the packet's Ethernet header is equal to the target host's MAC address) will be forwarded to Port #a; that is, to the malicious host.

As example of CAM table poisoning attack, Figure 1 shows that in the CAM table of a switch, there are four hosts connected to the switch. Host #1, the malicious host, attacks the switch's CAM table using 3 fake packets. The three packets are almost the same, but they have different source MAC addresses in the Ethernet headers. The information of the packets is as follows:

1. *First fake packet*: Source MAC address in the Ethernet header = 00:08:74:04:BC:4A (Host #2).
2. *Second fake packet*: Source MAC address in the Ethernet header = 00:08:74:05:AD:20 (Host #3).
3. *Third fake packet*: Source MAC address in the Ethernet header = 00:03:0D:38:79:57 (Host #4).

After this attack, the switch's CAM table becomes corrupted, as shown in Figure 2. The CAM table shows that all four hosts are connected to the switch's Port#1 (*FastEthernet 0/1*). However, physically only Host#1 is connected to Port#1.



Interface Description	Interface Type	Operational Status	Speed	MAC Address
FastEthernet0/1	ethernetCsmacd(6)	Up	100 Mbps	00:03:0D:38:79:57 00:08:74:04:BC:4A 00:08:74:05:AD:20 00:0F:1F:C0:EB:49
FastEthernet0/2	ethernetCsmacd(6)	Up	100 Mbps	
FastEthernet0/3	ethernetCsmacd(6)	Up	100 Mbps	
FastEthernet0/4	ethernetCsmacd(6)	Up	100 Mbps	

Figure 2: The content of the CAM table after the CAM table poisoning attack

Once a packet is sent to one of these three hosts (Host#2, Host#3 and Host#4), the switch will forward it to Port#1; that is, to Host#1. This situation may create a DoS situation, since the switch is not forwarding the packets, issued from these three hosts, to their destinations (Figure 3).

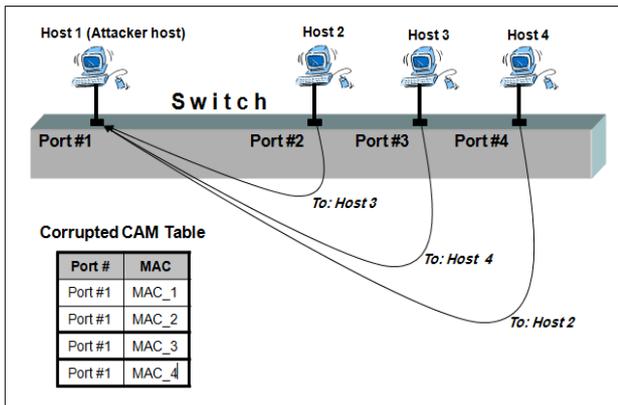


Figure 3: A DoS attack based on CAM table poisoning

3.2 Experiment

The following experiment describes how to poison the CAM table of a target switch. A simple network is used in the experiment. Three Windows XP based hosts are connected to a switch and each host is assigned a static IP address, as shown in Figure 4. The experiments discussed here use a switch device from a leader in the market namely Cisco, but the knowledge can be easily adapted to any other available switches with similar security features, such as Juniper switches.

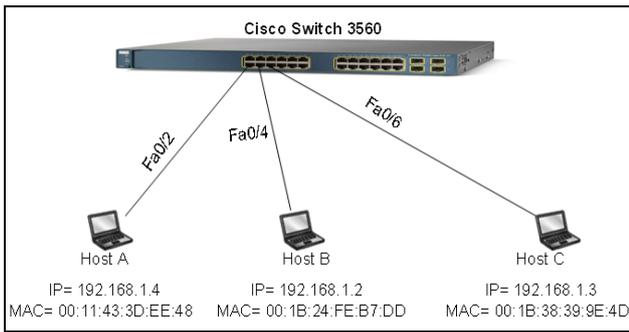


Figure 4: Network architecture

The experiment consists of the following two steps:

- Step 1: View the CAM table contents
- Step 2: Poison the CAM table contents

3.2.1 Step 1: View the CAM table contents

To view the CAM table contents of a switch, simply perform the following steps:

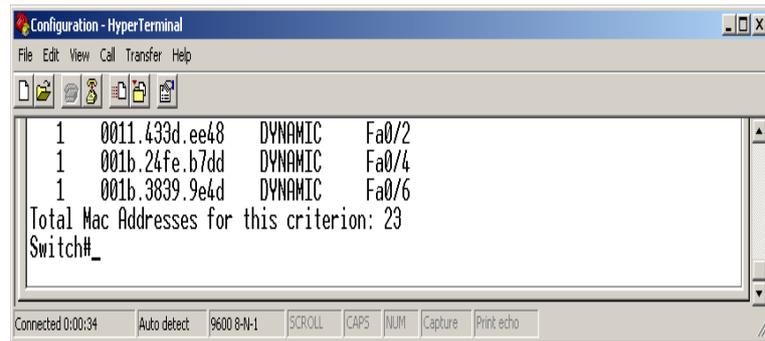
- Connect a LAN's host to the console port on the switch.
- Run the Terminal Application program (For example: HyperTerminal) in the host.
- Under "Connect Using:" option, select one of the appropriate communication port (COM1, COM2, etc.) that the console cable is attached.
- Select OK and a "Port Settings" window will pop-up prompting you to define the data rate and communication setting as defined by the vendor. (Most vendors have the following settings: 9600-Bits per second, 8-Data Bits, None-Parity, 1-Stop bits and None-Flow control.)
- Select OK. This will place you in the Terminal Window.

- Depress the "Enter" key a few times until a menu from the switch appears in the Terminal Window.
- If the menu appears, then you are ready to configure the switch as needed.
- In case of a Cisco switch (Cisco 2011), type the following command to view the contents of the CAM table:

Switch>enable //enter the enable command to access privileged EXEC mode.

Switch# show mac-address-table

- The CAM table content is:



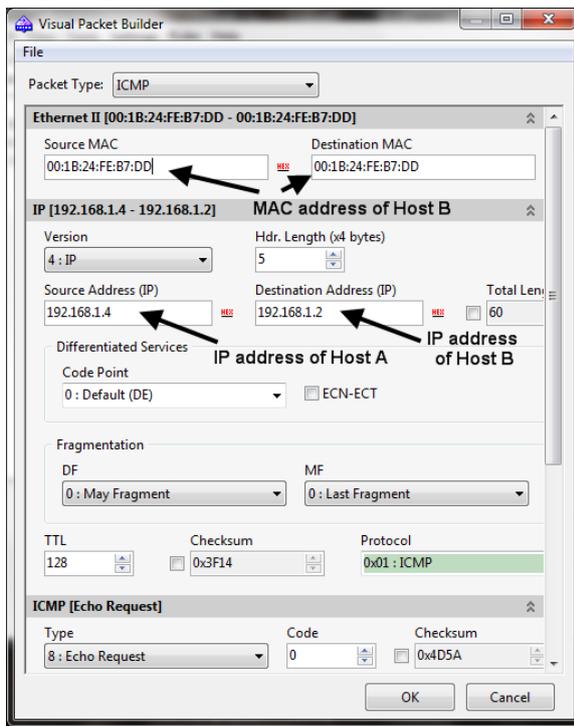
This screen shows that three hosts, whose MAC addresses are displayed, are connected on Port #2, Port #4, and Port #6, respectively.

3.2.2 Step 2: Poison the CAM table contents

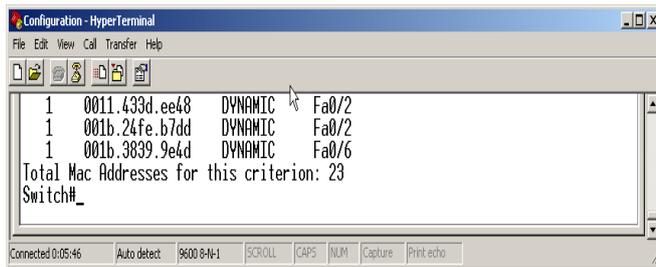
We assume that Host A wants to poison the switch's CAM table, by inserting the invalid entry: MAC address of Host B ↔ Switch's Port 0/2 (Fa0/2). This invalid entry will tell the switch that Host B is now located at Port 0/2 (Fa0/2). However, physically, Host B is still located at Port 0/4 (Fa0/4). Hence to perform this attack, Host A should send to any destination host in the LAN network a fake packet (IP or ARP packet) whose Ethernet source MAC address is equal to the MAC address of Host B:

IP or ARP packet	
Ethernet header:	
• Source MAC address	MAC address of Host B
• Destination MAC address	Any MAC address

Using any packet builder tool, such as *CommView Packet Builder* or *Engage Packet Builder*, the above fake packet can be easily built. In this lab exercise, we use *CommView Packet Builder*, since it provides a very friendly GUI interface to build IP, TCP, ICMP, UDP and ARP packets. For example, the following screenshot shows that a fake ICMP echo packet, whose MAC source is equal to the MAC address of Host B, is built at Host A:



After sending the fake ICMP echo packet, the CAM table becomes corrupted, as follows:



The above screenshot shows that Host B is connected on Port #2. However, physically, Host B is still connected on Port #4. Consequently, when a host in the LAN network sends packets to Host B, the switch will not forward them to Host B; in contrast they will be forward to Host A. This is a DoS attack, since the LAN network's hosts are not able to communicate properly with Host B.

3.3 MAC Flood Attack

An old attack technique for sniffing traffic in a switched LAN network is based on MAC flooding. MAC flooding is a technique employed to compromise the security of network switches. In a typical MAC flooding attack, a switch is flooded with many Ethernet frames, each containing different source MAC addresses, by the attacker. The intention is to consume the limited memory set aside in the switch to store the MAC address table. That is, some CAM tables of old switch models may be overflowed and revert to broadcast mode (hub mode known also as the 'fail open mode') as a consequence after which sniffing can be easily performed. After launching a successful MAC flooding attack, a malicious user could then use a packet analyser (a sniffer) to capture sensitive data being transmitted between other computers, which would not be accessible when the switch operates normally.

4 Lab exercise: Prevention of CAM Table Poisoning

This lab exercise is about preventing the poisoning of the switch's CAM table. The learning objective of this lab exercise is for students to learn how to protect switches from CAM table poisoning attack.

To prevent CAM table poisoning, security administrators usually rely on the presence of one or more features in their switches. With a feature often called "port security" by vendors, many advanced switches can be configured to limit the number of MAC addresses that can be learned on ports connected to end stations. A smaller table of "secure" MAC addresses is maintained in addition to (and as a subset to) the traditional CAM table.

For example, Cisco Catalyst 3560 Series switches (Cisco 2011) allow to restrict the number of legitimate MAC addresses on a port (or an interface) using the port security feature. When that number is exceeded, a security violation would be triggered and a violation action would be performed based on the mode configured on that port. Therefore, any unauthorized MAC addresses would be prevented from accessing and corrupting the CAM table.

A switch's port can be configured for one of three violation modes, based on the action to be taken if a violation occurs:

- *Protect*—when the number of secure MAC addresses reaches the maximum limit allowed on the port, packets with unknown source addresses are dropped until the switch administrator removes a sufficient number of secure MAC addresses. The switch administrator is not notified that a security violation has occurred.
- *Restrict*—when the number of secure MAC addresses reaches the maximum limit allowed on the port, packets with unknown source addresses are dropped until the switch administrator removes a sufficient number of secure MAC addresses. In this mode, the switch administrator is notified that a security violation has occurred.
- *Shutdown*—A port security violation causes the interface to shut down immediately. When a secure port is in the error-disabled state, the switch administrator can bring it out of this state by entering the *err disable recovery cause psecure_violation* global configuration command or he can manually re-enable it by entering the shutdown and no shut down interface configuration commands. This is the default mode. The switch administrator can also customize the time to recover from this state.

4.1 Experiment

The following experiment describes how to configure and test the port security feature in Cisco Catalyst 3560 Series switches to prevent the poisoning of the CAM table. The experiment uses the same network architecture described in the previous lab, and consists of the following steps:

- Step 1: Configure the Restrict Mode Security Port in the switch.
- Step 2: Generate a malicious packet to poison the CAM table.

- Step 3: Configure the Shutdown Mode Security Port in the switch.

4.1.1 Step 1: Configure the Restrict Mode Security Port in the switch

The following steps allow configuring the Restrict Mode Security Port:

- Connect a host to the console port on the switch
- Run the Terminal Application program in the host
- Type the following commands:

Switch>enable //enter the enable command to access privileged EXEC mode

Switch# Configure terminal

Switch(config)# interface fastethernet 0/2 // port security feature is applied on the host connected on Port #2

Switch(config-if)# switchport mode access

Switch(config-if)# switchport port-security

Switch(config-if)# switchport port-security violation restrict

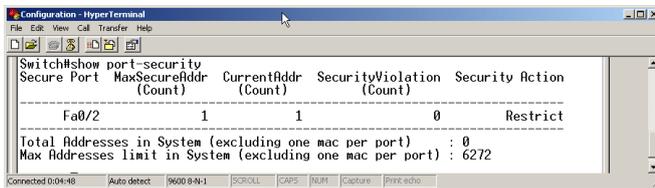
Switch(config-if)# end

Switch# copy running-config startup-config

- To display the port security mode, type the following command:

Switch# show port-security

- The following results will appear:

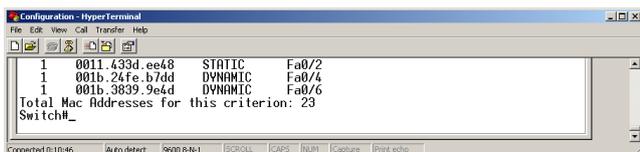


4.1.2 Step 2: Generate a malicious packet to poison the CAM table

Use any packet generator tool to generate a malicious packet whose MAC source in the Ethernet frame is equal to a fake MAC address. For example, we use the same fake ICMP echo packet generated in the previous hands-on lab exercise.

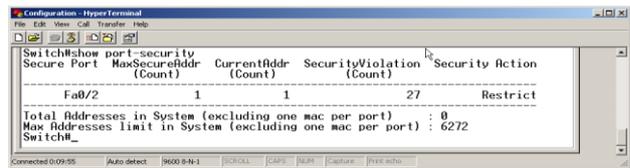
- Type the following command to view the CAM table contents after the poisoning attempt:

Switch# show mac-address-table



This screenshot shows clearly that the CAM table has not been corrupted.

- Display again the port security mode:



This screenshot shows that there have been 27 packets that attempted to violate the security feature implemented on Port #2. These packets attempted to corrupt the CAM table; however, the switch has blocked them.

4.1.3 Step 3: Configure the Shutdown Mode Security Port in the switch

Type the following commands to configure the Shutdown Mode Security Port:

Switch(config)# interface fastethernet 0/2

Switch(config-if)# switchport mode access

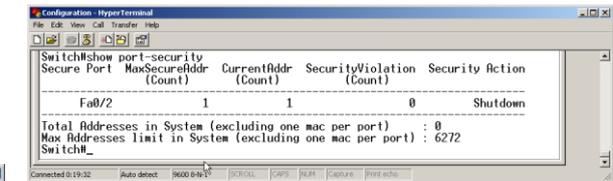
Switch(config-if)# switchport port-security

Switch(config-if)# switchport port-security violation shutdown

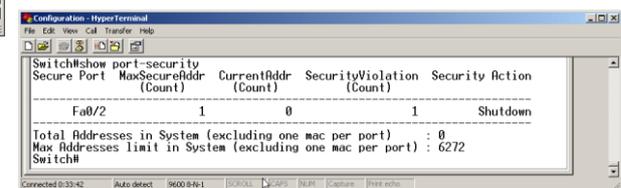
Switch(config-if)# end

Switch# copy running-config startup-config

- Display the port security mode:



- Generate the same fake ICMP packet of the previous test, and then display the port security mode:



This screen shows clearly that there has been a packet that attempted to violate the security feature implemented on Port #2. The switch has blocked the malicious packet and shut down the port.

The following screen shows clearly that Host A has lost its connection to the switch (Interface Fa0/2 has been shutdown), and a warning message appeared on Host A's desktop, as follow:



5 Ethical Concern

The hands-on lab exercises have been used in our intrusion detection and response course in the last three years. A major ethical concern has been identified when

analysing the number of malicious IP and ARP packets injected in the university network.

We used the intrusion detection sensors installed in the network segments to collect malicious packets and detect potential attack traffic. Figure 5 shows that the total average number of malicious packets targeting the university switches' CAM tables over the three years increased during the days following the hands-on lab exercises practice. This is a dilemma when offering hands-on lab exercises about offensive techniques.

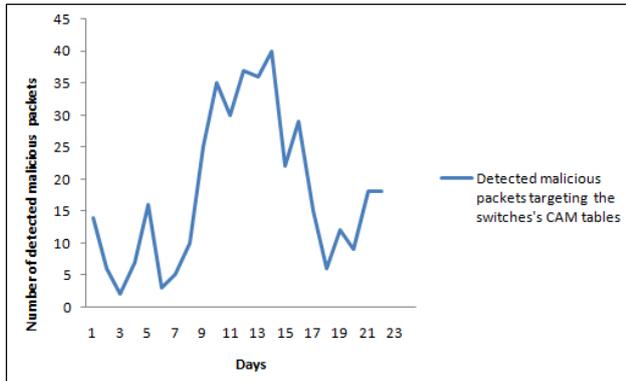


Figure 5: Evolution of the number of detected malicious IP and ARP packets targeting the switches' CAM tables

On the other hand, a survey showed that most of the students said that they have experiment the DoS attack using the CAM table poisoning technique, outside the university isolated network laboratory environment, particularly at their home's networks. The victims were mainly their sisters and brothers' computers. They used the DoS attack to prevent their victim computers from accessing the Internet. Table 1 shows the result of the survey conducted over the last three years on about 110 students enrolled in the intrusion detection and responses course.

Questions	Responses
Did you experiment the DoS attack, outside the university isolated network laboratory environment, after the hands-on lab exercises practice?	<ul style="list-style-type: none"> 82% of the students said "Yes" 13% of the students said "No" 5% abstained
If yes, where did you experiment the attacks?	<ul style="list-style-type: none"> At the home's network (72%) At the university's network (25%) At other networks (3%)
What were your objectives of attacking your victims?	<ul style="list-style-type: none"> For fun (76%) Deny the victim from accessing the Internet (24%)

Table 1: Student survey results (Number of students = 110)

It is often criticized that offensive methods should not be taught to students since this only increases the population of "malicious hackers". We feel that this line of argument is flawed. Any security technique can be simultaneously used and abused. The trend towards

penetration testing in corporate businesses shows that offensive techniques can be used to increase the level of security of an enterprise. So students trained in offensive techniques must not necessarily become black hats (malicious hackers), but rather can also become white hats (good security professionals). However, we agree that offensive techniques should not be taught in a standalone fashion. As with defensive techniques, every course in IT security should be accompanied by a basic discussion of legal implications and ethics. Students should be educated on their ethical responsibilities. Ethical behaviour is a mandatory part of information security curriculums.

6 Student's performance and satisfaction

From fall 2006 to spring 2008 (a two years period), students enrolled in the intrusion detection and response course were not offered hands-on lab exercises about CAM table poisoning attack technique. Only the conceptual part of the technique has been described in the class.

However, from fall 2008 to spring 2011 (a three years period), students were offered the hands-on lab exercises described in this paper. Over the last five years period, each semester the students were also given one quiz about switch's CAM table poisoning attack technique. Figure 6 shows the students total average grades for the quiz, per semester. It is clear that from fall 2008, the students' total average grade has started improving. This is mainly due to the fact that the hands-on lab exercises allowed students to better anatomize the attack technique and assimilate further the concepts learned from the lecture. The students have learned better with the hands-on lab exercises which had a positive effect on their grading performance.

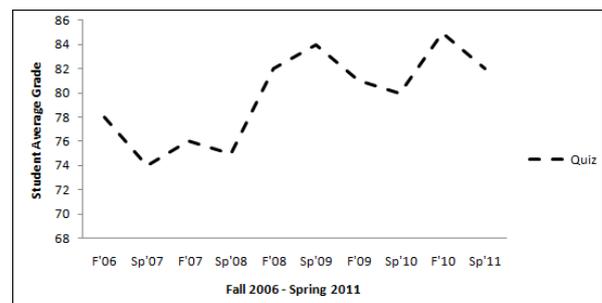


Figure 6: Student total average grades in the quiz

On the other hand, the students were given a questionnaire survey to assess their overall satisfaction with the hands-on labs and get their feedback. The student survey results are listed in Table 2. Overall the students' feedback was positive.

Questions	Responses
Did you enjoy the labs?	<ul style="list-style-type: none"> • 87% strongly agree • 10% agree • 2% neither agree or disagree • 1% disagree
Do you think the labs are easy to follow and straightforward?	<ul style="list-style-type: none"> • 82% strongly agree • 10% agree • 5% neither agree or disagree • 3% disagree
Do you feel you understand the concepts better after performing the labs?	<ul style="list-style-type: none"> • 85% strongly agree • 13% agree • 1% neither agree or disagree • 1% disagree
How likely are you to recommend the labs to others?	<ul style="list-style-type: none"> • 86% strongly agree • 11% agree • 2% neither agree or disagree • 1% disagree
Would you like to see these labs (or similar labs) used in your network security classes?	<ul style="list-style-type: none"> • 87% strongly agree • 8% agree • 4% neither agree or disagree • 1% disagree
Laboratory exercises helped me to learn how to apply security principles and tools in practice.	<ul style="list-style-type: none"> • 85% strongly agree • 8% agree • 5% neither agree or disagree • 2% disagree

Table 2: Student survey results (Number of students = 40)

7 Conclusion

This paper described in detail two hands-on lab exercises. The first hands-on lab exercise is about how to perform practically DoS attack using switch's CAM table poisoning. The second hands-on lab exercise is about the implementation of "Security port" feature available in common switches for preventing the attack. The two hands-on lab exercises allow students to better anatomize and elaborate the attack in an isolated network laboratory environment. They are designed to be used as a part of an undergraduate-level course on network security and intrusion detection and prevention course.

However, a major ethical concern has been identified when analysing the alert logs generated by the intrusion detection sensors installed in the university networks. This is a dilemma when security students are exposed to offensive hands-on lab exercises. However, the ethical concerns of teaching students "hacking" are dwarfed by the need for knowledgeable, competent, and, above all, experienced computer security professionals in industry and government.

8 References

Harris, J. (2004): Maintaining ethical standards for computer security curriculum. *Proc. of the 1st Annual Conference on Information Security Curriculum Development*, NY, USA, 46-48, ACM Press.
 SwitchSniffer:

<http://switchsniffer.en.softonic.com/>. Accessed 26 Oct 2011.
 Winarp: <http://www.arp-sk.org/>. Accessed 25 Oct 2011.
 WinArpAttacker: http://www.mobile-download.net/Soft/Soft_2641.htm/. Accessed 20 October 2011.
 Cisco Systems, Catalyst 3560 Series Switch Cisco IOS Software Configuration Guide: <http://www.cisco.com/>. Accessed 20 Oct 2011.
 Mink, M. and Freiling, F. (2006): Is Attack Better Than Defense? Teaching Information Security the Right Way, *Proc. of the 3rd Annual Conference on Information Security Curriculum Development*, Kennesaw, Georgia, USA, 44-48, ACM Press.
 Arce, I. and McGraw, G. (2004): Guest Editors' introduction: Why attacking systems is a good idea. *IEEE Security & Privacy*. 2(4):17-19.
 Arnett, K. P. and Schmidt, M. B. (2005): Busting the ghost in the machine. *Communications of the ACM*, 48(8):92-95.
 Dornseif, M., Holz, T. and Mink, M. (2005): An offensive approach to teaching information security: *Aachen Summer School Applied IT Security*. Technical Report AIB-2005-02, RWTH Aachen.
 Vigna, G. (2003): Teaching network security through live exercises. *Proc. of the Third Annual World Conference on Information Security Education (WISE 3)* 3-18, Monterey, CA, USA, Kluwer Academic Publishers.
 Yuan, D. and Zhong, J. (2008): A lab implementation of SYN flood attack and defense. *Proc. of the 9th ACM SIGITE conference on Information Technology Education, SIGITE '08*, Cincinnati, Ohio, USA, 57-58, ACM Press.
 Caltagirone, S., Ortman, P., Melton, S., Manz, D., King, K. and Oman, P. (2006): Design and implementation of a multi-use attack-defend computer security lab. *Proc. of the 39th Annual Hawaii International Conference on System Sciences*, Hawaii, USA, 9:220c.
 Bishop, M. (1997): The state of INFOSEC education in academia: Present and future directions. *Proc. of the National Colloquium on Information System Security Education*, 19-33.
 Frincke, D. (2003): Who watches the security educators? *IEEE Security & Privacy*. 1(3): 56-58.
 Hill, J., Carver, C., Humphries, J. and Pooch, U. (2001): Using an isolated network laboratory to teach advanced networks and security. *Proc. of the 32nd SIGCSE Technical Symposium on Computer Science Education*, Charlotte, North Carolina, USA, 36-40, ACM Press.
 Mullins, P., Wolfe, J., Fry, M., Wynters, E., Calhoun, W., Montante, R. and Oblitey, W. (2002): Panel on integrating security concepts into existing computer courses. *Proc. of the 33rd SIGCSE Technical Symposium on Computer Science Education*, NY, USA, 34(1), 365-366, ACM Press.
 Yuan, X., Matthews, D., Wright O., Xu, J., and Yu, H. (2010): Laboratory Exercises for Wireless Network Attacks and Defenses. *Proc. of the 14th Colloquium for Information Systems Security Education*. Baltimore, Maryland, USA, 116-123.
 Livermore J. (2007): What are faculty attitudes toward teaching ethical hacking and penetration testing? *Proc. of the 11th Colloquium for Information Systems Security Education*, Boston, MA, USA, 111-116.

Implementation of a Smart Lab for Teachers of Novice Programmers

Ali Alammary, Angela Carbone and Judy Sheard

Faculty of Information Technology

Monash University, Melbourne, Australia

Asala3@monash.edu.au, angela.carbone@monash.edu, judy.sheard@monash.edu

Abstract

Communication between students and their instructors in the lab is a limited commodity. With limited access to the tutor, students can sometimes spend a long time trying to fix simple errors, continually revisiting and repeating the same errors. Instructors, on the other hand, find themselves explaining the same mistakes over and over again. It is often not clear to them how well individual students are progressing toward meeting the task objectives. This paper introduces a new implementation of Smart Classroom technology for introductory programming computer laboratories. The Smart Lab is intended to make the computer lab a better educational environment for both students and instructors. In the Smart Lab instructors are provided with information about each student's progress as they perform programming tasks, enabling the instructors to readily respond to individual student's problems and assess the overall progress of the class. Two different evaluation approaches were used to test the new implementation: an expert review session and a lab study. The evaluation found that the Smart Lab improved instructors understanding of their students' problems enabling them to provide timely and appropriate feedback. It also provided instructors with better understanding of their students' programming strategies and compilation behaviours.

Keywords: Smart Classroom, introductory programming, learning technology, tutoring, feedback.

1 Introduction

Learning programming is a difficult task for most students. Novice programming students experience many problems which contribute to high dropout and failure rates in introductory programming courses (Lister et al., 2004; McCracken et al., 2001). According to Gomes and Mendes (2007), these problems start in the early stages of learning when students are trying to understand and apply basic programming concepts, such as loops and control statements. A common argument used to explain the difficulties students face is that programming is a multifaceted skill. Learning to program requires knowledge about programming languages, programming

tools and problem solving (Robins, Rountree and Rountree, 2003; Carbone, Mitchell and Hurst, 2009).

One way to assist students overcome difficulties in learning to program is by providing them with programming tasks to solve in the computer lab with support from a tutor. A tutor can inspect the students' code, investigate the problem and then provide the students with the appropriate help. However, communication between students and their instructors in the lab is a limited commodity. As the number of students in the lab increases, the amount of time that instructor can devote to each student decreases. With limited access to the tutor, students can sometimes spend a long time trying to fix simple errors, continually revisiting and repeating the same errors. They may have difficulty understanding the compiler messages, making code correction a frustrating experience. Instructors, on the other hand, find themselves repeatedly explaining the same mistake. It is often not clear to them how well individual students are progressing toward meeting the task objectives. Furthermore, they are not easily able to assess the errors that students are most commonly encountering.

A number of tools have been developed to assist instructors in giving feedback to students about their work. All tools found in a search of the literature were designed to provide feedback about students' assignments after they had completed and submitted their work, and not for their work on programming activities in lab classes. In view of this, we have developed a *Smart Lab* that is intended to improve the assistance instructors provide to their students in their lab classes. It allows tutors to better understand their students' problems, programming strategies and compilation behaviours. As a result, students can be provided with appropriate help when they need it and common mistakes can be realised and explained.

This paper investigates a Smart Classroom technology for introductory programming computer laboratories, implemented as a Smart Lab. Section 2 details the Smart Lab development, with features of a new Smart Lab described in section 3. In section 4 details of the evaluation of Smart Lab are provided, followed by a discussion, conclusion and suggestions for future work.

2 Related work

As background to the development of the Smart Lab we have investigated two areas: i) implementations of Smart Lab technology ii) approaches to helping address the difficulties faced by students when working on a programming task.

2.1 Smart Lab technology

A Smart Lab can be defined as a lab equipped with tools designed to enhance instruction and learning (Di, Gang and Juhong, 2008). The idea behind the Smart Lab is to employ technology to provide students and teachers with tools to extend their ability to communicate effectively and to enable them to engage successfully with the curriculum. A Smart Lab can be implemented in different ways to achieve different goals. As an example, Tissenbaum and Slotta (2009) describe a lab which was developed in several layers using a variety of approaches and devices. The main goal of that implementation was to enable students to visualize problems and their solutions on a large display screen in the lab. This visualization was intended to help students connect and understand the relationships between these problems and their solutions.

The most common implementation of smart lab technology is for distance learning and is concerned with making distance education an exciting experience and as effective as face-to-face instruction. A notable example of this implementation is the work of Di et al. (2008) who presented what they called the blending-reality classroom. This system was designed to provide the instructors with intellectualized human-computer interaction technology for their teaching purposes, and also to provide the distance education students with an appropriate study environment.

In general, the different implementations of smart lab technology found in the literature were complex and contained many different components. They also required expensive hardware such as video cameras, servers and display screens. Of the smart lab technology reviewed, none were found to deal specifically with a computer programming lab, rather they were designed to work in any kind of lab or classroom.

2.2 Approaches to address the difficulties faced by programming students

Many studies have been conducted to address the difficulties faced by programming students when working on programming tasks (Carbone, 2007; Sheard et al., 2009). These studies can be classified into three areas: 1) identification of the difficulties encountered by novice programmers, 2) provision of automated help to students via their development platforms, and 3) provision of tools that help instructors monitor their students' progress. The motivation for much of this work is to reduce the workload of instructors and to enhance students' learning experiences. The following provides examples of studies from these three areas.

Many studies have investigated the difficulties encountered by novice programmers. For example, Jadud (2006) studied the ways in which novice programmers write their programs. He examined which errors were most often generated by novices and the time they would spend to fix these. Another work by Flowers, Carver and Jackson (2004) captured and explained syntax and semantic errors made by programming students. They determined the fifty most common programming errors and found that even the 'stronger' students continued to make these fifty errors late into the semester. The main reason for this, according to Flowers et al, is that the Java

compiler error messages do not help students to understand and fix their errors. These studies have provided valuable information about the obstacles students face in writing programs and the approaches they take to overcome these.

Another body of work has focused on development of tools to provide automated help to students. An example is the web-based application *WebToTeach* developed by Arnow and Barshay (2002) that enables automated checking of students' assignments. *WebToTeach* provides students with a list of programming tasks written by their tutor. Students write their answers to each exercise in a dedicated form and then wait for the system to provide feedback. If the answer passes a certain number of system checks, the student will be informed by the system that no extra work is needed. Otherwise, the answer will be rejected and the student will be provided with an explanation and hints to correct their submission. Another example is the tool *Espresso*, which is targeted at students studying introductory Java programming. *Espresso* provides students with easy to read error messages (Hristova et al., 2003). *Espresso* does not eliminate the need for compiler messages; rather it enhances the compiler functionalities by generating detailed and easy to read error messages and providing suggestions on how to fix the errors.

A further body of work has focused on development of tools to enable instructors to monitor their students' progress. Spacco et al. (2006) developed a tool called *Marmoset* that provides instructors with detailed feedback on the development process of their students' programming assignments. Snapshots of students' code are captured and stored on a central repository each time students save their files. This process enables generation of development histories for each student. These histories offer a detailed perspective of students' progress while doing a programming task. Another tool *Retina* (Murphy et al., 2009) does a similar job as *Marmoset* but focuses more on the compilation and run time errors. *Retina* collects data about students' programming activities and stores these in a central database. It then processes this data to provide instructors with detailed information about students' progress. Karam, Awad and Carbone (2010) developed a tool to analyse students' code and extract what they termed *actions*. Each single statement or declaration in the program is considered as an action. After extracting the actions from the student code, the tool extracts actions from the solution code provided by the instructor. It finally compares the actions from the student's code to those from the solution code, to generate a list with the missing and completed actions.

The tools that were found to assist instructors have been developed to investigate student code away from a class situation, for example, their assignment work. However, the number of assignments that students do in a semester is relatively small compared to the number of tasks they attempt in the programming lab sessions. Despite this, no tools were found that could be used in a teaching situation in a computer lab. Furthermore, except for the last mentioned tool, all the other tools rely on the code's output and the compilation attempts to evaluate the students' code. Such evaluation is insufficient to provide comprehensive insight into students'

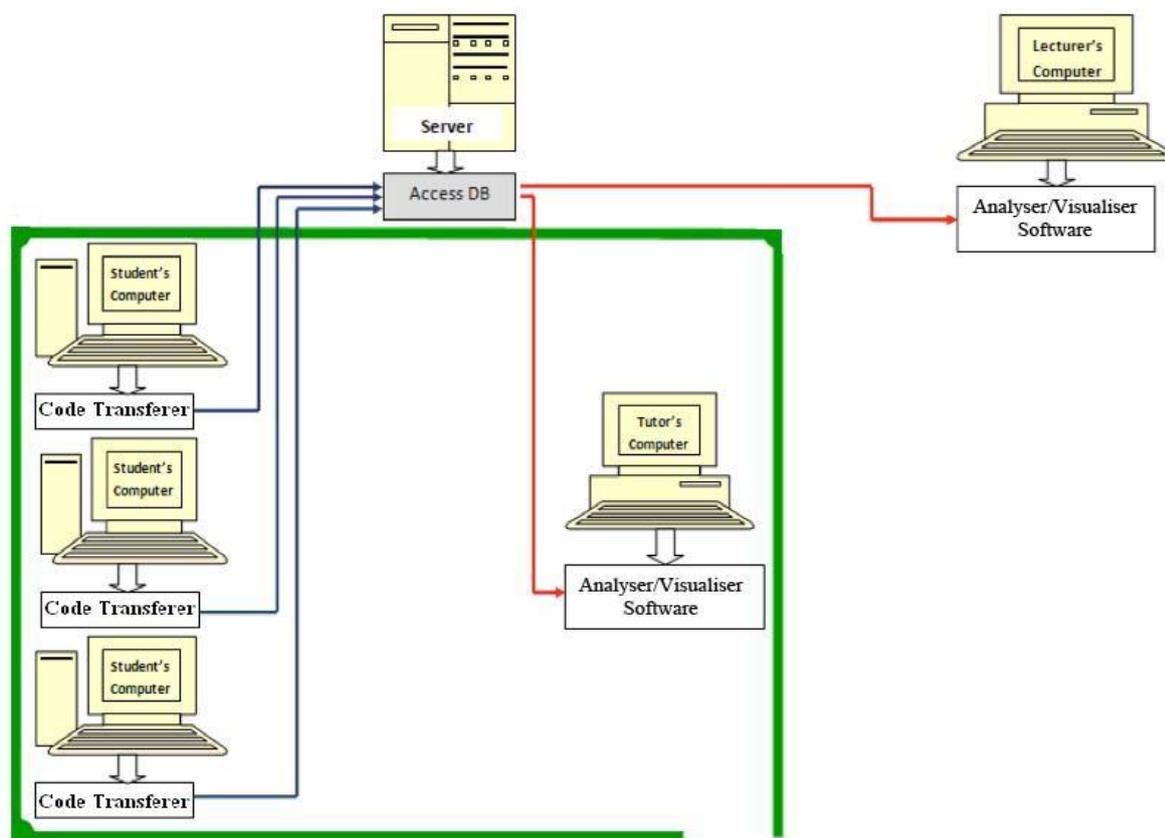


Figure 1: The Smart Lab architecture

programming behaviors and the different types of difficulties they experience.

Our work fills this gap by developing a *Smart Lab* implementation which incorporates and builds upon the code analysis tool of Karam, Awad and Carbone (2010). The Smart Lab is essentially a tool for use in a computer lab which provides instructors with detailed information about each student's progress in writing computer programs.

3 Our Smart Lab solution

Our Smart Lab is a computer lab containing a set of networked workstations equipped with specially designed software. Code on student machines is captured, analysed and a summary displayed on the instructor's machine. The instructor is thus able to monitor progress of each student in the class from one central point.

The Smart Lab was designed to be simple and easy to use. It does not require any extra hardware and can be easily installed on any workstation in a lab. It needs to be configured one time only in the lab. All configuration setting is stored in a single text file called *BlueJSetting*. To configure the workstation, this file can be easily copied to the C drive on the workstation. The architecture of the Smart Lab is shown in Figure 1. It analyses the compilation attempts and the code output, and then further analyses the code to provide deeper insights into the students' problems and programming

behaviour. There are two main components to the Smart Lab: the *Code Transferer* component, used to transmit real-time information about the students' code and their programming activities to a central database, and the *Analyser/Visualiser* software component, that interacts with the central database to provide information to the instructor on the students' progress and the problems they experience. Each of these is explained in detail in the sections that follow.

3.1 Code Transferer

The aim of the *Code Transferer* is to send real-time snapshots of students' code and their programming activities to the central database. Currently, the *Code Transferer* is associated with the BlueJ IDE, but it can be easily modified to work with other IDEs. When the Code Transferer starts, it registers three types of listeners: *Package Listener*, *Compile Listener* and *Class Listener*.

Package listener does two main tasks: it captures a snapshot of the students' code and detects the current class that students are working on at the moment and stores this information in the database. *Compile Listener* in turn captures and stores detailed information about both successful and unsuccessful compiling attempts. The *Class Listener* is used to detect the changes to the state of each of the classes that students are implementing. The state of the class changes when the class has been opened in the editor, compiled, uncompiled or renamed.

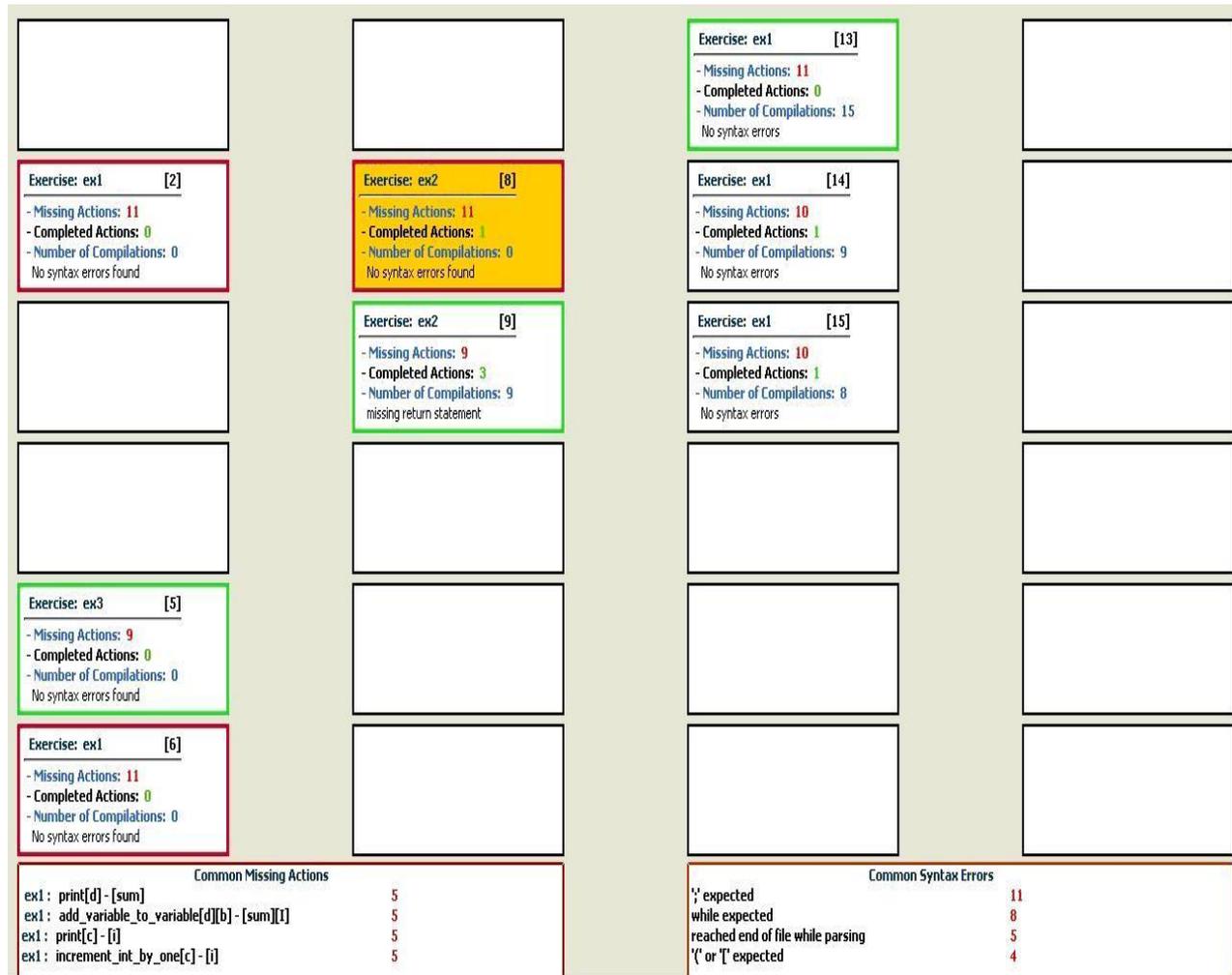


Figure 2: The Analyser/ Visualiser main window

3.2 Analyser/Visualiser

The *Analyser/Visualiser* allows tutors to monitor their students' progress in real-time when they are working on programming tasks in the lab. It also allows other teaching staff to access students' captured data at a later stage to view the different strategies that students used to implement solutions, as well as the common difficulties they encountered. These features can help tutors to understand students' needs and provide each of them with more appropriate and focused assistance. Lecturers can also utilise this feature to identify topics and concepts that students are experiencing problems with and tailor their instruction to those areas.

The *Analyser/Visualiser* offers seven different features. These include:

1. Graphical representation of the lab setting
2. Common Missing Action box
3. Common Syntactic Errors box
4. Highlighting which students require the most and the least help
5. Highlighting students who remain idle for extended periods

6. Detailed information about a particular student's activities
7. Students' final reports

Each of these features is explained in detail in the remainder of this section.

3.2.1 Graphical representation of the lab

Each student in the lab is represented by a label in the *Analyser/Visualiser's* main window (Figure 2). To enable easy mapping of each label to each student, the position of each label on the screen is related to the actual location of the student in the lab. Each label contains a summary of the student's activities. This includes the current exercise that the student is working on, the number of successfully implemented actions, the number of actions not yet implemented, the number of compilation attempts of the current exercise and the result of the last compilation attempt.

The missing and implemented actions are identified by using the tool developed by Karam, Awad and Carbone (2010) which was described in the related work section. The graphical representation is designed to help the tutor readily observe the amount of progress that each student

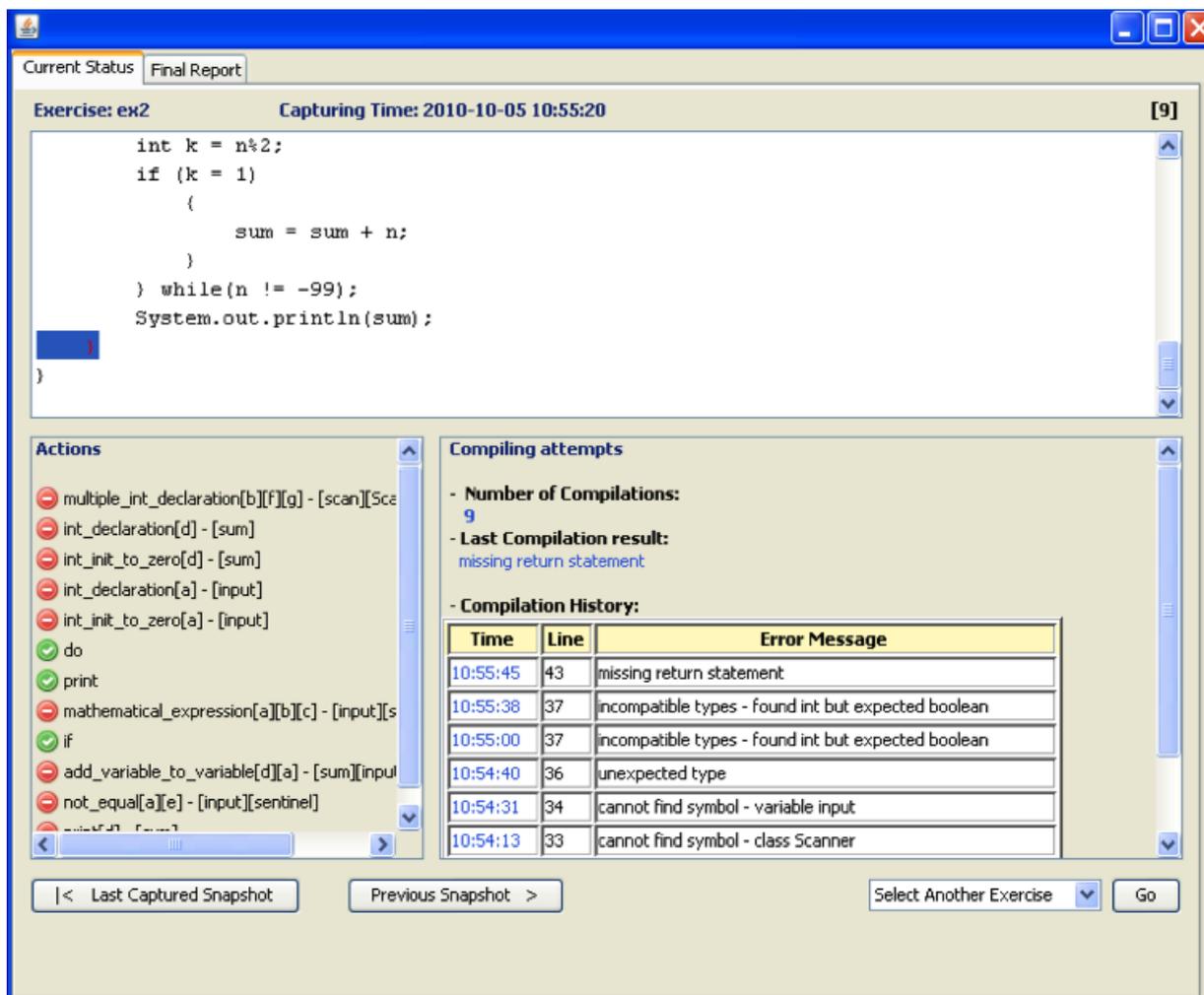


Figure 3: Detailed window with detailed information on a student’s activities.

has achieved in solving a programming exercise, understand the strategy that each student used to solve the programming exercise, and identify the exercise that took students the longest time to implement.

3.2.2 Common Missing Actions box

This box appears at the bottom left-hand corner of the *Analyser/Visualiser* main window and it shows the four most common missing actions that the students failed to implement (see Figure 2). This box is meant to help the tutor identify the actions that students find difficult to implement and to identify programming concepts that require additional clarification.

3.2.3 Common Syntactic Errors box

This box appears at the bottom right-hand corner of the *Analyser/Visualiser* main window and it highlights the four most common syntactic errors that the students made (see Figure 2). This box is meant to aid the identification of common syntactic errors that students make, thereby enabling the tutor to focus more on these errors and to advise the students how to avoid them.

3.2.4 Highlighting students who require the most and the least help

The *Analyser/Visualiser* changes the border colour of the labels that represent the students to red for the students that need most help and to green for the students who need the least help. This highlighting is meant to help tutors focus teaching efforts on the students in the lab according to the level of help they may need. Figure 2 indicates that students 2, 6 and 8 need the most help while students 5 and 9 need the least help. Note that the main window shows the missing actions for the current exercise only, while the application looks to the whole missing actions across all exercises to determine which students need the most or least help.

3.2.5 Highlighting students who remain idle for extended periods

The *Analyser/Visualiser* software changes the background colour of the labels to orange if a student has remained idle for more than two minutes. Highlighting the idle students is meant to provide tutors with more insight into students’ programming behaviours and they

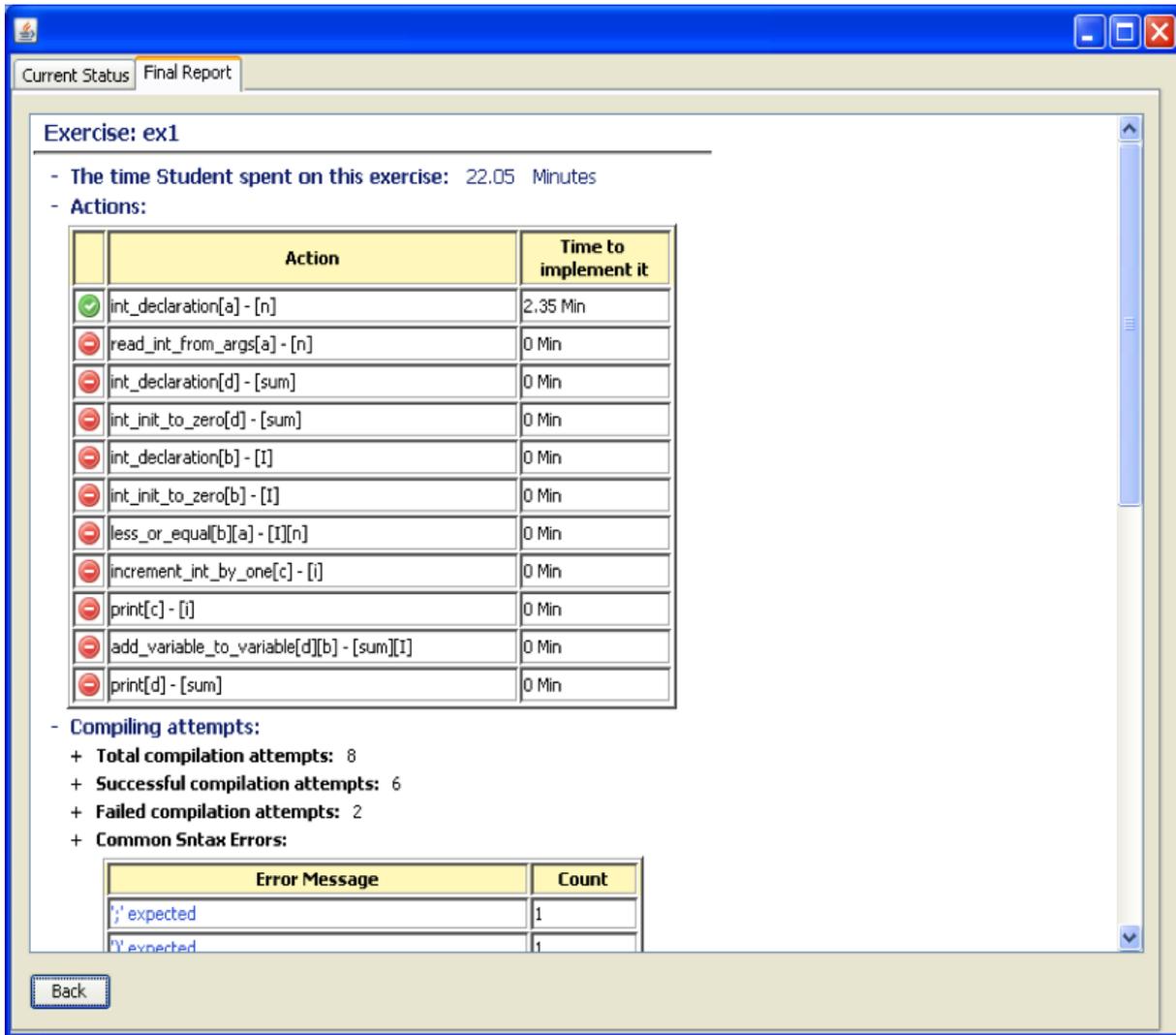


Figure 4: Student's final report.

may then investigate the source of their inactivity. Figure 2 shows an example of this functionality; the system has changed the background colour of student 8 who remained idle for a period of time.

3.2.6 Detailed information about a particular student's activities

Clicking on a particular student's label in the main window opens a new window with detailed information about that student's programming activities (see Figure 3). This window contains: the name of the current exercise the student is implementing, the last captured snapshot of code, a list of completed and missing actions and compilation attempts for that class. The window also allows the instructor to navigate forwards and backwards through the code snapshots by clicking the navigation buttons at the bottom of the window. It also has a drop-down list to allow tutors to navigate to the other exercises that the student has already attempted.

The information in this window sheds additional light on the student's progress. It pinpoints the nature of the assistance that the student requires, tracks the amount of progress that a student is making in the task and aids understanding of the student's programming strategy.

3.2.7 Student's final report

The final report (see Figure 4) shows the amount of time the student spent on each exercise, the final analysis of the student code and detailed information on the compilation attempts. It provides an indication of the amount of effort that student has put into solving each programming task, the programming concepts that the student had problems with and his/her compilation behaviour.

4 Evaluation approach

The evaluation of the Smart Lab solution combined two evaluation techniques: an expert review session and a lab study. This approach of using two evaluation techniques is recommended by Krug, Burghardt and Edwards (2002) to provide a better assessment of computer systems rather than just using one technique.

4.1 Expert review

Four lecturers who teach programming units in the Faculty of Information Technology at our University were invited to evaluate the Smart Lab system in an

expert review session. Each lecturer had had many years of experience in teaching introductory programming units both in lecturing and tutoring in lab situations. During the session, each participant had the opportunity to take on the role as a tutor and use the *Analyser/Visualiser* software to monitor the programming activities of other participants who act as students working on programming exercises. At the end of the session each participant was interviewed on their experiences of using the Smart Lab system.

The evaluation questions were divided into three different sections, with each section focusing on a different part of the Smart Lab interface.

1. **Analyser/Visualiser main window.** Questions in this section were designed to investigate: 1) the usefulness of the information in the main window and whether it provided sufficient understanding of the programming activities of the other participants, 2) the ease of the mapping between the labels representing the other participants and their actual location in the lab, and 3) the helpfulness of highlighting the students who required the most and the least help.
2. **Detailed window** (see Figure 3). Questions in this section investigated the participants' opinions about whether the information in the detailed window provided an understanding of the programming strategies and compilation behaviour of the other participants and whether it was helpful in spotting the different kinds of problems encountered by the participants performing the programming tasks.
3. **Students' final reports** (see Figure 4). Questions in this section were designed to investigate whether the report information was helpful in identifying the amount of effort each participant put into solving the programming tasks and whether the report information provided an understanding of the programming strategies and compilation behaviour of the other participants.

4.2 Lab Study

Novice programming students in an introductory programming unit taught within a Masters program in the Faculty of Information Technology at our University were invited to participate in this study. The study was conducted in two laboratory classes each with a different tutor and different groups of students. In each session the tutor used the Smart Lab system to monitor their students' programming activities. At the end of each laboratory session the tutors were interviewed about the system.

The interview questions were divided into three sections:

1. The purpose of this section was to solicit the tutor's reactions to the Smart Lab system and gather their opinions of the positive and negative aspects of the Smart Lab system.
2. This section explored the tutor's experiences while using the Smart Lab system and with each of the Smart Lab features.

3. In this section the tutor was asked to rate the usefulness of the system. He/she was then asked to discuss the benefits of the system and suggestions for improvement.

5 Results and discussion

In the expert review session, all four lecturers reported that the *Analyser/Visualiser* software provided them with useful information about the programming activities of the other participants who were playing the role of students. They all also agreed that it helped them to identify the participants who were not progressing on their programming task and to understand their compilation behaviour and programming strategies. Three lecturers claimed that it was easy for them to do mental mapping between the labels representing the other participants and their actual location in the lab, and to identify the different kinds of problems they were facing.

In the lab study, the tutors stated that the graphical representation of the lab helped them identify the students' programming activities. They agreed that the missing actions and syntax error boxes were useful in spotting the common problems that students encountered. The colour change in the labels and their borders also helped the tutor identify which students needed the most help. However, a couple reported that analysis of the missing and completed actions did not accurately reflect the students' actual actions. This identified a problem with the Karam, Awad and Carbone (2010) tool used for the analysis.

The tutors also agreed that the students' detailed window helped them to understand the students' programming strategies and the problems encountered by each student. They found the final report useful in helping them to identify the amount of effort that individual students put into each task.

On a 10 point scale, where 1= useless and 10= very useful, the tutors both gave the system a 9 rating. One commented that "we might have rated it even higher if we had been able to use it for longer time".

Overall, the experts and tutors found the Smart Lab system a valuable tool to helping instructors identify the common problems that the participants encountered with their code. They also found that the system enabled them gain a better understanding of the programming difficulties that the participants were having with their code. The tutors from the study lab commented that each student's problems were easy to identify.

The Smart Lab system also seemed to provide enough information to the instructor to effectively help the students when the need arose. Both experts and tutors agreed that, based on the information provided by the system, they could better prioritize their time to help those students in need of most help. They were also able to provide the right type of help.

The experts and tutors agreed that the system is potentially very useful for gaining understanding of the students' programming strategies and compilation behaviour. They found the detailed information that appeared when clicking on a particular helpful them in fully understanding the students' programming and compiling strategies.

Finally, both experts and tutors agreed that the information in the final report was of great help in discovering the progress made by each student towards completing the exercise.

6 Future work and conclusion

This study produced the first implementation of a smart classroom technology for the introductory programming units' labs. The implementation of the Smart Lab system was successful in achieving its intended aim of providing instructors with specific, timely and detailed information about their students' performance on programming tasks during the lab class sessions. The Smart Lab facilitated the communication between the tutors and their students in the lab, and improved the overall effectiveness of the lab session in achieving the learning objectives. Results indicate that it is a useful aid for laboratory instructors by helping them:

- find the common problems that students faced with their code;
- address their students' difficulties when the need arose;
- understand their students' programming strategies and compiling behaviours;
- identify the progress made by each student in solving the programming exercises.

This study has opened up new research areas for further improvements and future work. The Smart Lab system could be extended in three different ways:

1. by allowing the tutor to provide online help and feedback directly to the students in their development platform. This saves the need to always go to the students' machines or disturb the whole class by explaining a problem on the board, as it the case with the normal programming labs.
2. by providing automated help to the students about the errors they make and the actions they miss. The feedback could come in the form of hints. These hints would be generated as easy to read messages, based on the actions students have not yet implemented and their syntax errors.
1. by discovering whether use of such monitoring tool actually has an impact on students' behaviour. The fact that there is someone looking over your shoulder could be inhibiting. The impact might even be positive - take a little more time to think about each error rather than repeating the tinker/compile cycle every fifteen seconds.

7 References

- Arnaw, D. & Barshay, O. (2002): WebToTeach: an interactive focused programming exercise system. *29th ASEWIEEE Frontiers in Education Conference* San Juan, Puerto Rico, 12A9, IEEE.
- Carbone, A. (2007): Principles for designing programming tasks: How the nature of task characteristics influence students learning of programming. PhD Dissertation, Monash University.
- Carbone, A., Mitchell, I., & Hurst, J. (2009): An exploration of internal factors influencing student learning of programming. *Proceedings of the 11th Australasian Computing Education Conference (ACE2009)*, Wellington, New Zealand.
- Di, C., Gang, Z., & Juhong, X. (2008): An introduction to the technology of blending-reality smart classroom. *2008 International Symposium on Knowledge Acquisition and Modeling*, Wuhan, 516-519, IEEE Computer Society.
- BlueJ - Teaching Java - Learning Java. Retrieved August 2010 from <http://www.bluej.org>.
- Flowers, T., Carver, C., & Jackson, J. (2004): Empowering students and building confidence in novice programmers through Gauntlet. *34th ASEE/IEEE Frontiers in Education conference*, T3H-10 - T3H-13, IEEE.
- Gomes, A. & Mendes, A. J. (2007). Problem solving in programming: *19th Annual Workshop of the Psychology of Programming Interest Group (PPIG2007)*, 216-228.
- Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003): Identifying and correcting Java programming errors for introductory computer science students. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, 153-156, ACM Press.
- Jadud, M. C. (2006): Methods and tools for exploring novice compilation behaviour. *Proceedings of the second International Workshop on Computing Education Research*, Canterbury, United Kingdom, 73 – 84, ACM Press.
- Karam, M., Awa, M., Carbone, A., & Dargham, J. (2010): Assisting Students with Typical Programming Errors During a Coding Session, *Seventh International Conference on Information Technology*, Las Vegas, Nevada, USA, 42-47.
- Krug, K., Burghardt, D., & Edwardes, A. (2002): *Usability Testing Template, D2.2.3*. WebPark.
- Lister, R., S. Adams, E., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., et al. (2004): A multi-national study of reading and tracing skills in novice programmers. *In Working Group Reports from Innovation and Technology in Computer Science Education (ITiCSE-WGR '04)*, 117-150.
- McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.-D., et al. (2001): A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students. *Working Group Reports from Innovation and Technology in Computer Science Education (ITiCSE-WGR '01)*, SIGCSE Bulletin, 125-140.
- Murphy, C., Kaiser, G., Loveland, K., & Sahar, H. (2009): Retina: Helping Students and Instructors Based on Observed Programming Activities. *Proceedings of the 40th ACM SIGCSE Technical Symposium on Computer Science Education*, Chattanooga, TN, USA, 178-182, ACM Press.
- Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J. K., & PaduaPerez, N. (2006): Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. *Proceedings of the 11th Annual*

SIGCSE Conference on Innovation and Technology in Computer Science Education, Bologna, Italy, 13-17, ACM.

Robins, A., Rountree, J., & Rountree, N. (2003): Learning and Teaching Programming: A Review and Discussion. *Journal of Computer Science Education*, 13 (2), 137-172.

Sheard, J., Simon, M. Hamilton, and J. Lönnberg, Analysis of research into the teaching and learning of programming. *Proceedings of International Workshop on Computing Education (ICER 2009)*, Berkeley, California, USA, 2009, 93-104.

Tissenbaum, M., & Slotta, J. D. (2009). A new framework for smart classroom research: Co-designing curriculum, research and technology. *Proceedings of 9th International Conference on Computer Supported Collaborative Learning*, Rhodes, Greece: International Society of the Learning Sciences, 91-93.

Evaluation of an Intelligent Tutoring System used for Teaching RAD in a Database Environment

Silviu Risco, Jim Reye

Computer Science Discipline, Faculty of Science and Technology
 Queensland University of Technology (QUT),
 GPO Box 2434, Brisbane, QLD 4001,
 Email: {s.risco, j.reye}@qut.edu.au

Abstract

This paper presents an evaluation of the Personal Access Tutor (PAT), an Intelligent Tutoring System (ITS) for Learning Rapid Application Development (RAD) in a database environment. We first give an overview of Microsoft Access, the environment that PAT uses. After describing related work in the field, we discuss the architecture of PAT and the services that PAT offers to the students, together with a short introduction of how students use PAT. After presenting the evaluation methodology, the results of a summative evaluation are discussed. Additional evaluation using data gathered from students by PAT is analysed as a pre-post test. The paper concludes with a summary and describes further work.

Keywords: ITS evaluation, Intelligent Tutoring Systems, Student Modelling.

1 Introduction

1.1 Microsoft Access overview

Microsoft Access (aka Microsoft Office Access) is a Relational Database Management System (RDBMS) developed by Microsoft Corporation. From this point on, to simplify the text, Microsoft Access will be called Access. Access is the most widely used Windows desktop RDBMS.

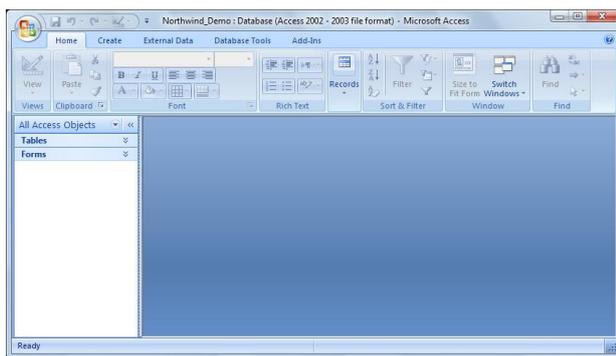


Figure 1: Microsoft Access - graphical interface

Several versions of Access have been developed by Microsoft Corporation. The latest version is Access

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the Fourteenth Australasian Computing Education Conference (ACE2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 123, Michael de Raadt and Angela Carbone, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

2010. PAT was initially developed for Access 2003 and later ported to Access 2007 and 2010. PAT can now be installed on any of the combinations of Access 2003 - Access 2010 and Windows XP, Vista or Windows 7.

Access is based on the Microsoft Jet Database Engine and provides a graphical user interface to create and use databases - see Figure 1. Through the graphical interface, the users can create several types of objects such as forms and reports to easily interrogate or update the database. Figure 2 and 3 show examples of forms and reports created in Access.

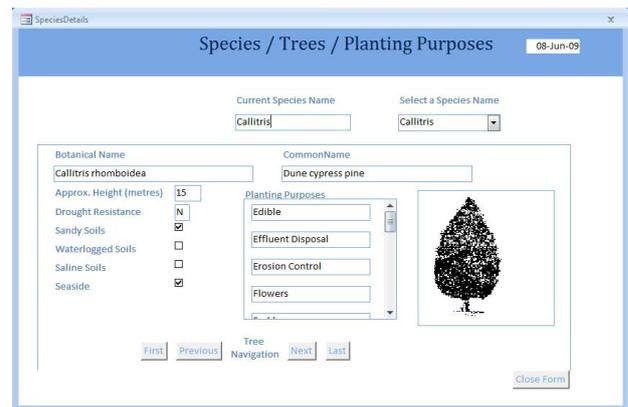


Figure 2: Microsoft Access - example of a form

Forms are objects used to enter, view or edit records in a database; reports are formatted printouts of the content of one or more tables or queries from a database (Adamski & Finnegan 2008).



Figure 3: Microsoft Access - example of a report

Access users can use built-in wizards to create simple

forms and reports, but these have restricted layouts and functionality. While using the wizards provides an easy start for users who have just started to learn Access, the wizards cannot be used to automatically create more complex and more advanced forms and reports. Such forms and reports must be manually created, although the wizards can be used to create a basic version, as a starting point. Users can also design forms and reports from scratch.

1.2 Teaching RAD using Access

“Databases” is a first year subject at Queensland University of Technology (QUT). While the first part of the subject covers SQL, the second part provides the opportunity to use a commercial RDBMS for Rapid Application Development (RAD) while applying knowledge learned about SQL. Because it is a widely used RDBMS, students learn how to use Access. Students have weekly practicals where they are required to solve exercises related to that week’s topic.

For the Access part, students learn how to create queries, forms and reports. First, they have to create simple forms and reports using the wizards, then they are required to improve the initial forms and reports, adding more functionality and improving their appearance.

1.3 Using PAT as an additional tool

PAT, an ITS for Access, aids students’ learning, complementing the lectures and practicals. PAT is freely available for QUT students. Students can use PAT both during the practicals or at home, in their own time and place.

It can be used to learn how to create forms and reports in Access, using PAT’s built-in exercises. It can also be used for assistance when working on assignments for the Databases subject.

1.4 Related work

Although of a limited number, other ITSs for learning about databases exist. These ITSs focus on teaching database domains such as Structured Query Language (SQL) and Database Design. SQL is a database language program designed for data management and manipulation for relational database management systems. Database Design is the process of creating a model of the information that will be held in the database. In this section we briefly describe some of these ITSs.

DB-suite (Mitrovic et al. 2004, Mitrovic et al. 2008) consist of three web-based intelligent tutoring systems in the area of databases:

- SQL-Tutor: teaches the SQL query language;
- NORMIT: a data normalization tutor; and
- KERMIT: teaches conceptual database modelling using the ER model.

These tutors are constraint-based tutors (Mitrovic & Ohlsson 1999, Mitrovic & Weerasinghe 2009, Ohlsson 1992). In the case of constraint-based tutors, the system analyses the student’s solution, checking if any constraints from the domain model are violated. The constraints are both for correctness and completeness. If a solution does not violate any constraints, then the solution is considered correct and complete.

The DB-suite tutors are designed to be used as an additional tool, to complement classroom teaching.

SQL Tutor (Mitrovic 1998, Mitrovic & team 2008) is a constraint-based ITS for students learning SQL. When using the tutor, the students have to complete SQL statements satisfying the given requirements.

The system contains definitions of several databases and a set of problems, together with their ideal solutions. The domain model of SQL Tutor contains more than 700 constraints.

Kermit (Suraweera & Mitrovic 2002, 2004) is an ITS for teaching Database Design using the Entity-Relationship (ER) data model. The students have to create an ER diagram based on the requirements given by the system. Kermit provides feedback to the students by request only.

The students can ask for a hint or can ask for the solution to be evaluated.

The feedback level is automatically increased each time the student asks for help, up to the *hint* level. Kermit contains over 200 constraints, both syntactic and semantic constraints.

Normit (Mitrovic 2002) is an ITS for students learning Database Normalisation. Normalisation is part of the database design. Data normalization is concerned with data optimisation, to minimise redundancy. Because Database Normalisation is a procedural task, the students have to follow a strict sequence of steps to solve the problem and the system does not need to store a correct solution.

The domain model of NORMIT contains more than 80 constraints (both syntactic and semantic) to check the student’s solution.

The hints have only two levels: a general hint and a more detailed hint. On the first time of violating a constraint, the system presents the general hint. When the rules are violated again, the more detailed hint is presented.

Acharya (Bhagat et al. 2002) is a web-based ITS for learning SQL. Acharya only analyses SQL for database querying, not updating.

This ITS uses Java servlet technology on a web-based front-end and PostgreSQL as a back-end.

Acharya contains a student module and a pedagogical module. The architecture has two separate databases, one for the student model, and the other one for the rest of the models - including the problems and their solutions. The student model contains general information about the student, history of concepts learned, with a confidence factor (the system’s belief that the student acquired the concept), knowledge level and number of hints received.

In contrast to SQL Tutor, Acharya uses a real RDBMS to run the students’ solutions and the result of the query is returned back to the student - if the query is correct. However, the students must still use the tutor’s interface to write the parts of the select statement.

Acharya stores in its student model general information about the student and history of information about the concepts learned (Bhagat et al. 2002). The concepts learned are recorded with a *certainty factor* which is a measure of Acharya’s belief that the student has acquired the concept. In addition, Acharya also records a knowledge level and the number of hints asked by the student.

Acharya can propose problems to the student based on pre-requisite relations. If the student’s solution is correct, the result of the SQL is displayed. If the student’s solution has errors, the most basic ones are addressed.

Showing the results of the query the students created as returned by a real RDBMS is very beneficial for the students as they can see exactly the result of their work.

SQL Lightweight Tutoring Module (SQL-LTM) (Dollinger 2010) is a system that can provide semantic feedback on SQL statements, pointing out their logic flows, even if they are syntactically correct. It can detect most conceptual errors that SQL learners can make (Dollinger 2010).

SQL-LTM is integrated with a Web based AJAX universal query tool called AJAX Enabled Query (AEQ) (Dollinger et al. 2009).

SQL-LTM consists of two modules: a query parser which converts the SQL query into an XML representation, and an analyser which compares the test query provided by the student against the reference query created by the instructor - also provided in an XML representation.

One of the difficult issues in analysing SQL queries is the possibility that the students can provide solutions that even though are different than the optimal solution, they can still be syntactically and semantically correct. The analyser recognises the semantic equivalence of such queries and provides recommendations on how to get to the expected solution.

Similar to Acharya, SQL-LTM uses a real RDBMS to run the queries from the students' solution. However, SQL-LTM does not have a student model nor does it keep track of student's history; hence the system cannot individualise the feedback. For the same error, different students receive the same advice.

2 Personal Access Tutor (PAT)

2.1 Architecture and components

An ITS architecture usually contains a simulation module which is used to replicate the real environment that the student is learning about. The architecture of PAT is different from this because it uses the real working environment (Access) instead of a simulation module.

Figure 4¹ presents the architecture and main components of PAT.

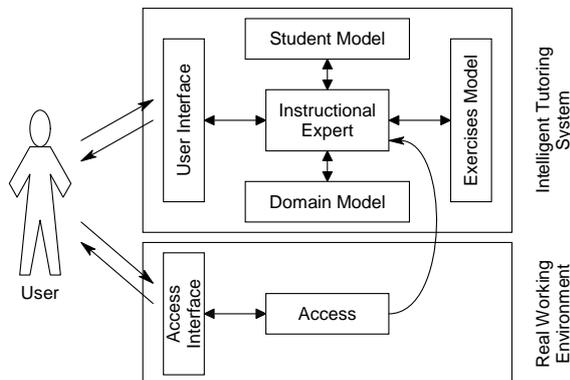


Figure 4: PAT's architecture

The *Domain Model* represents the knowledge about the domain to be taught, knowledge that an expert in the field should know. It is the foundation for the entire knowledge base. Because PAT's focus is on helping students to learn how to create forms and reports, all the objects (together with their properties) that can be created in a form or a report are present in the *Domain Model*.

¹based on the general architecture of an ITS presented by Burns & Parlett (1991)

The *Student Model* contains the system's beliefs regarding the student's knowledge of the domain (Holt et al. 1994) and additional information about the student, such as personal characteristics and learning style (Beck et al. 1996). In PAT, the Student Model includes information about student's preferences for learning from diagrams or text, their interests in the subject and in several topics from the domain. Every time PAT analyses the student's solution, the Student Model is updated with new information.

The *Instructional Expert*, based on knowledge both about the domain and the student, diagnoses the student's attempted solution and provides individualised feedback. As part of the Instructional Expert, the Tutoring Model contains information about teaching the domain such as tutoring goals and hints for students. The Tutoring Model must be able to take advantage of the information provided from the Student Model (e.g. student's learning style and personal characteristics). The Instructional Expert in PAT is based on principles from the Minimalist Framework for designing instructional materials for computer users (Carroll 1990); the GOMS model (Card et al. 1983); and Andragogy, "the art and science of helping adults learn" (Knowles 1980, p. 43) - a student centred approach for adults.

Access / Access Interface: because PAT is implemented as an add-in for Access, the student can utilise PAT from within Access. After installation, PAT appears as a new group in the Access's ribbon. In this way, the student can actually work on each exercise, test their solution and receive feedback from PAT without leaving Access's main window.

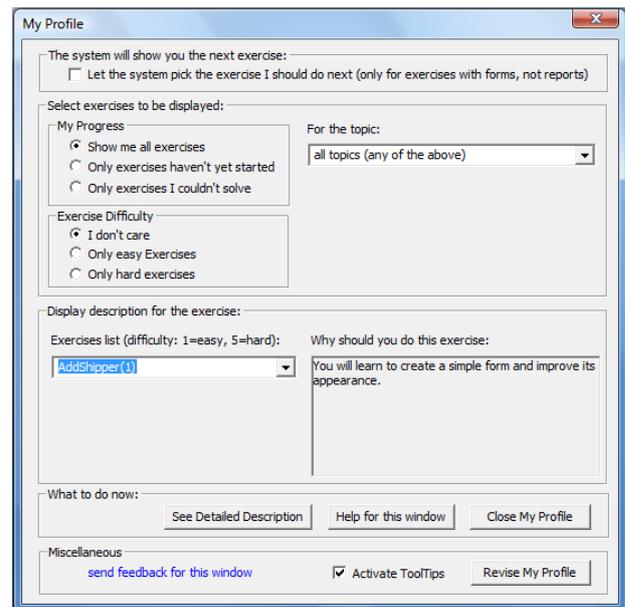


Figure 5: PAT's My Profile window

2.2 Services that PAT offers

VanLehn (2006) presents a global approach to ITSs behaviour. Based on the concepts of *Task* (a multi-minute activity that can be skipped or interchanged with other tasks) and *Step* (multiple user interface events that together can complete a task), an ITS is presented as having 2 loops: the outer and the inner loop. The *outer loop* is responsible for the task selection, similar to the "elaborative function" identified by Self (1987). The *inner loop* consists of the steps inside the task: assessment of knowledge (diagnostic

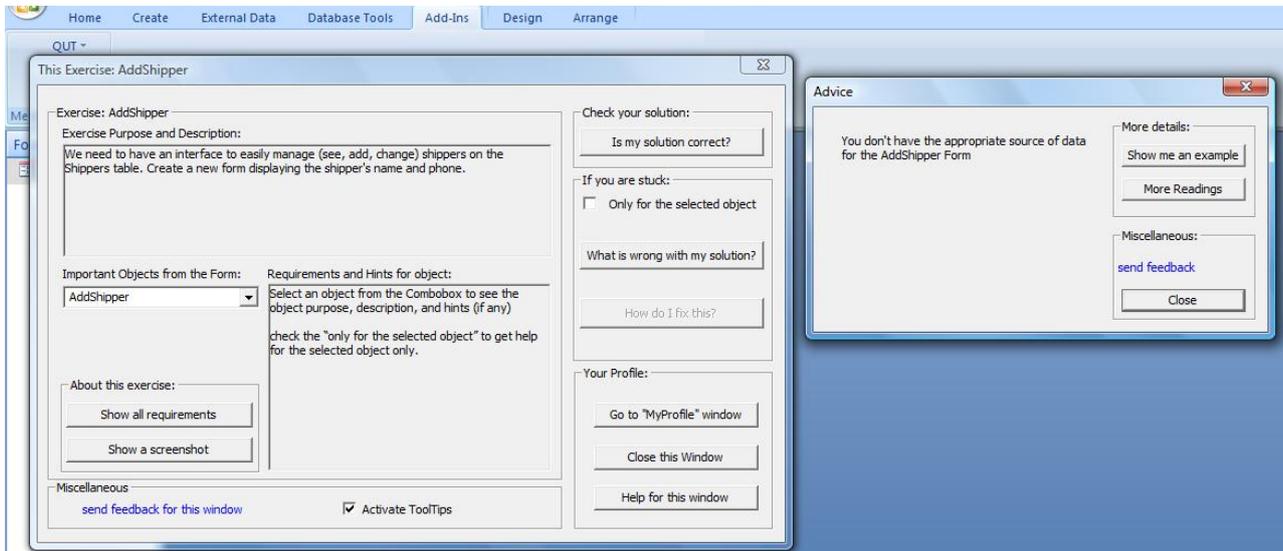


Figure 6: PAT's This Exercise window

function), feedback and hints (corrective function), etc.

PAT's *outer loop* (task selection) gives the student two choices: the student can select an exercise or can ask PAT to suggest the next exercise (or the first exercise, if the student did not try any exercises yet). The selection can be narrowed down by specifying the difficulty of the exercises or whether the exercise was previously attempted. The exercises are categorised by topics (and subtopics) to be learned and contain (beside the exercise description) the exercise difficulty and what the user will achieve by completing the exercise. Figure 5 shows the *MyProfile* window, which is the interface from where the user can choose which exercise to do next.

PAT can propose an exercise based on the information from the student model and considering the following principles:

- The exercises should contain topics not mastered yet by the student.
- The topics not known yet should help to complete broader teaching concepts.

PAT's *inner loop* relates to the steps within a task and can be grouped in two main categories of services offered to the user: step generation and step analysis. While the step generator is about what the user should do next, the step analyser is responsible for other actions such as answering the question "Is it correct?", and other types of feedback. In PAT, the user can access these services from the *ThisExercise* window (left hand side of Figure 6).

- Is my solution correct?
- What is wrong with my solution?
- How do I fix this?

Is my solution correct? gives the user an overall presentation of what is correct and what is incorrect with their solution. PAT displays the "Traffic Lights" image (Figure 7) where there is a row for each important task in the exercise. A green light means the task is correctly done, while a red light means the task is completely wrong or missing. Because a task can consist of a series of steps, a yellow light means that *only some* steps are correct, not *all* of them.

What is wrong with my solution? gives minimal feedback - it only describes what the error is. *How do I fix this?* gives the user feedback about the actions that should be done to correct an error. This type of feedback is not enabled the first time the user asks for help. Generally, the user has to first choose *What is wrong with my solution?*, and only after that can they ask for help on how to fix the error. In the case where PAT detects that the user will not benefit from receiving a general (vague) hint, a more specific hint will be provided. That could be about how to fix the error, rather than what is wrong.

The feedback for the last two services above is grouped on several levels of specificity, starting from a general hint and leading to more specific hints. However, not even the most specific feedback gives away the correct solution because PAT can be used even for assignments.

In addition to the services described above, PAT can display a diagram depicting the context of the error (Figure 8). PAT can also give users references to readings related to the topic where the error occurred. The additional readings are from lecture notes, recommended books or lecture slides.

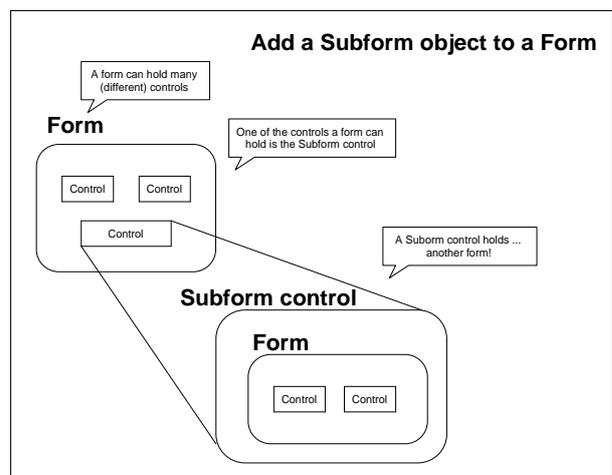


Figure 8: Example of a diagram that PAT displays

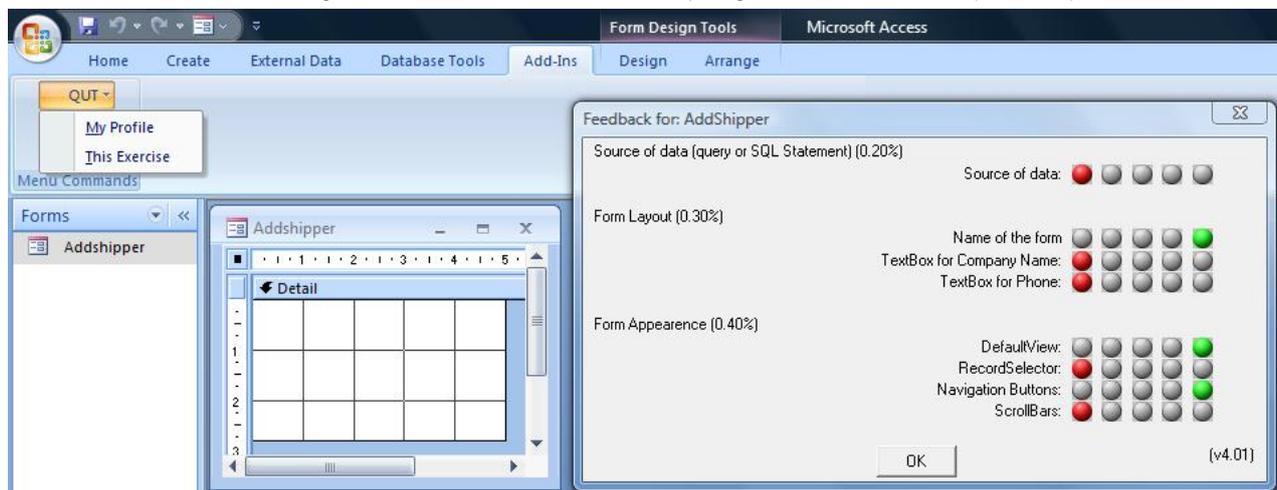


Figure 7: PAT's This Exercise window

3 How PAT is used

PAT is freely available to QUT students enrolled in the Database subject and can be downloaded from the learning content management used at QUT. PAT can be installed on home computers or laptops, as well in computer labs. However, to take maximum advantage of the Student Model, installing PAT on students' laptops or home computers is recommended.

There are two ways in which PAT can be used:

- go through enough exercises to cover (and master) the entire curriculum; and
- use the system mainly to work through the assignments.

The first option is more suitable for students who do not have any previous knowledge working with Access or other RDBMS. The second option is preferred by students with some previous knowledge with Access or students with less interest in the subject. The two options of using PAT are presented next.

3.1 Best way to work with PAT

The best way to work with PAT is to take full advantage of the Student Model that PAT will update during the student-system interaction. This implies both following the exercises that PAT will propose and also keeping the Student Model. While the first only needs the student to ask PAT for the next exercise instead of manually select one, the second requires the student to consistently use the same installation of PAT - same computer. In this way, every exercise that the student will attempt will be recorded in the Student Model.

As the student works on the exercise that PAT proposes, (s)he can check if their solution is correct. By displaying the traffic lights, the student has a good indication of what (if anything) is wrong with their solution. If the error is not obvious, the student can ask PAT what is wrong.

Before asking for more help, the student should first check the helping diagrams and more readings section, trying to discover by themselves what to do next. Only if the student is still stuck after more readings, should they ask PAT for hints on how to fix the problem.

3.2 Using PAT for assignments

PAT is released to the students at the same time as the assignments. Because there are three Access assignments, there are three releases of PAT, which include (in addition to the existing exercises) the requirements and solutions for the assignments. The solutions are kept hidden from student view, via encryption and other security mechanisms.

Students' previous experience with Access or other RDBMSs can vary significantly. We understand that some students have already used Access. Therefore assuming that they will go through the entire process described in the previous section would be wrong. In these circumstances, we allow the students a different approach. They can try to use only the *Is my solution correct* feature, and keep working without spending time on more readings - assuming that they have the required knowledge.

4 Initial evaluation of PAT

Evaluations were conducted with students who used PAT. A survey was used to provide both qualitative and quantitative data.

Students enrolled in this subject have diverse backgrounds and their previous experience with Access (if any) varies significantly. For an accurate interpretation of students' answers, we distinguish between students who have used Access before and those who haven't. Another important aspect that has to be considered is how much they used PAT during the semester. If some students only seldomly used PAT they will not be in the same category as students who used PAT extensively.

4.1 Objectives and Methodology

Iqbal et al. (1999) suggested that the evaluation method for an ITS should be chosen by what is being evaluated (the entire system or only a part of the system) and the number of available students. We wanted to analyse PAT as a system and we had 185 students enrolled in the Databases subject in second semester 2008. We used questionnaires as an exploratory research method. During one of the lectures, the students present in the lecture theatre were asked to fill in a two pages questionnaire. Of 185 students enrolled in the subject, 84 responded to the

questionnaire. However, only 51 of the students answered all questions.

The objectives for the evaluation were:

1. Get students' backgrounds.
2. Does PAT's architecture (receiving help when solving real problems) make learning easier?
3. Is the feedback provided helpful for students? Is PAT offering enough types of help?
4. Is PAT accepted by students?

4.2 Conclusions for Summative Evaluation

This section describes the summative evaluation of PAT. We present the results from the evaluation with students, then the results from the evaluation with teaching staff. The results from both students and teaching staff are summarised below, grouped by the evaluation's objective. Because objective 4 measures the overall rating of PAT, it is presented first.

Objective 4 - Is PAT accepted by students and teaching staff?

As shown in Figure 9, we can see that both students and staff found PAT helpful. Furthermore, from questions 9 and 10 for students and question 6 for staff we see:

- the majority of students and staff members think that PAT is easy to use; and
- students would like to have software similar to PAT in other subjects.

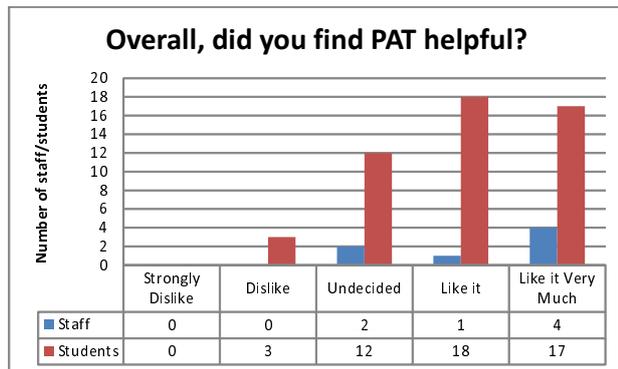


Figure 9: Students and teaching staff answers for questions 11 and 7.

Objective 1 - Students and staff members' background

Two thirds of the students enrolled in the subject have used Access before, with more than 50% of them being *Somewhat Confident* in using it. Interestingly, the answers to the questions between the two groups (students that used Access before and students that didn't use Access before) are similar in the majority of the questions.

The question where the results are different are:

- Q7 - Which type of feedback did you like least?
- Q11 - Overall, did you find PAT helpful?

Students who used Access before were not really interested in additional materials such as *Diagrams*, *More Readings* and not even in *How to Fix* while students who haven't use Access before did not like *More Readings* and *What is Wrong* but were happy with *Diagrams* and *How to Fix*.

Although one might expect that PAT would be more useful for beginners (students who had not used Access before), the results show that 56% of them answered that they like or like very much PAT and 38% answered that they are undecided. In contrast, 76% of the students that used Access before answered that they like or like very much PAT with only 18% undecided.

Objective 2 - Does the approach of having PAT embedded in Access make learning easier?

Both students and teaching staff considered that using PAT directly from Access, while using the real software (no simulation) to work on real problems, is very useful. Students' answers for question 10 can also be seen as a confirmation - the students would like to have software similar to PAT in other subjects.

Objective 3 - Is the feedback provided helpful for students? Does PAT offer enough types of help?

Questions 6 and 7 show students' preferences for the types of feedback provided. The *Traffic Lights* (error indication) are by far the most liked type of feedback see Figure 10. While the students who used Access before were looking more for a simple way of indicating what is wrong, the students who didn't use Access before were looking not only for an indication of "what is wrong" but also for an indication on the overall performance and progress.

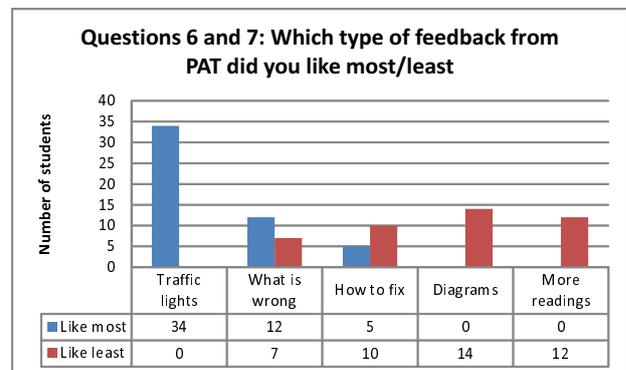


Figure 10: Type of feedback that students like most/least.

The least preferred types of feedback are *More Readings* and *Diagrams*: *More Readings* for the students who hadn't used Access before, while *Diagrams* for the students who had used Access before. Some teaching staff and students did not indicate any type of feedback as disliked (Question 7 for students and Question 4 for staff): "N/A - no particular dislike" or simply "none". The only suggestion for improvements from staff members (Question 3) was to not only have general diagrams describing the overall concept but also screen-shots from Access on how to solve some of the possible issues.

Some students were unhappy with the content of the feedback received. One possible explanation is the fact that PAT only gives hints, not the solution (correct answer) for assignments. However, future work could look at ways to improve the feedback.

5 Pre-Post test evaluation

In addition to the evaluation described in the previous section, we gathered measurements of the students' knowledge before and after using PAT. During the interaction between the student and PAT, PAT records every click on any of its interfaces, together with additional information about students solution

at that time such as: the current error, the advice given, etc. The information is recorded in another Access database containing the student model. When submitting the assignment, the students were asked to also submit this database containing their student model.

Because PAT helps with the assignments, we could not ask any of the students not to use PAT just to have a control group. For this reason, we evaluated PAT using the approach described by Woolf (2008, p. 191) as “C1. Tutor alone”. This evaluation was conducted during the second semester of 2010, with a different set of students.

5.1 Information sources

To analyse the improvement engendered in students’ learning while using PAT we have two available sources of information: data gathered by PAT during student-system interactions and the students’ solutions to the assignments.

During each interaction between the student and the system, PAT will record not just the current session, the error that was addressed, the advice type, advice code, etc. (as part of the student model) but also the name of the dialog box and the button - any time the student clicks on one of those. The data is recorded in the database containing the student model. The information in this database allows us to see which topics were not initially understood (not known) by students while they were using PAT - either practicing on the helping exercises or working on the assignments.

Additionally, the students’ solutions to the assignment provides us with information not just about the topics not understood by the students (not known), but also the topics shown as understood (known) by the students.

The first source of information (student-system interaction) was used to provide pre-test data, while the information from the second source (the assignments) was used to provide post-test data.

5.2 The student population

Of the 235 students enrolled in the Databases subject in the second semester 2010, 199 students submitted all the assignments and student models. Because using PAT was optional, the information received from some of these 199 students was insufficient for an accurate evaluation of the data for those students.

Possible criteria for selecting the relevant students (students that used PAT enough to provide useful data) are:

- number of interactions (clicks on PAT’s interfaces) - a maximum of 1598 and a mean of 182.5;
- number of advice messages received - a maximum of 394 and a mean of 30; and
- number of sessions started - a maximum 151 and a mean of 11.

The first criterion above is a good measure of how much a student used PAT, in order to distinguish between significant and insignificant data. This criterion was used to determine the best set of data - that for the first 100 students, in descending order of their numbers of interactions.

5.3 The topics considered

From the 38 topics (object-properties) existing in the exercises used during both the pre and post tests, we selected the 10 most relevant ones based on the following criteria:

- the topic should require the student to set a correct value (i.e. not topics that can be easily generated by the wizards or using default values); and
- the topic should be important from a teaching perspective (i.e. some object-properties are more important than others).

5.4 Results of the pre-post evaluation

In Section 5.1, we explained that the data being used for pre-test purposes provides information about topics not known initially. Where students did not need help, we take it for granted that they already knew the topic. The post test data provides information about topics known and unknown.

From this data, we obtained the average number of topics known on the pre-test versus average number of topics known on the post-test. These values are shown in Figure 11.

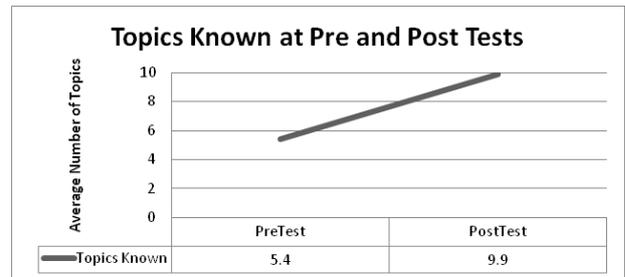


Figure 11: Topics known at pre and post tests.

On average, the number of topics known increased from 5.4 to 9.9, after using PAT, i.e. the average number of topics learned is 4.5. The mode is 4 topics learned, with a standard deviation of 1.87. Woolf (2008, p. 191) lists for a “tutor alone” evaluation the questions that should be addressed:

- Do learners with high or low prior knowledge benefit more?
- Do help messages lead to a better performance?

To address the first question, we analysed the pre-post results for students with prior knowledge of Access versus students with no prior knowledge of Access. When a student starts using PAT for the first time, it asks the student if they are confident with using Access. The answer is recorded in the student model.

Based on this information, 55 students (out of 100 students analysed) had prior knowledge of Access - i.e. they answered “yes” to the question if they are confident with using Access. The results of the pre-post test for the two categories of students are shown in Figure 12

It can be seen from the graph that the students with no prior knowledge had a slight increase in the number of topics learned in comparison with students with prior knowledge of access. Students with no prior knowledge had an average of almost 6 topics learned compared with 5 topics learned by students with prior knowledge of access.

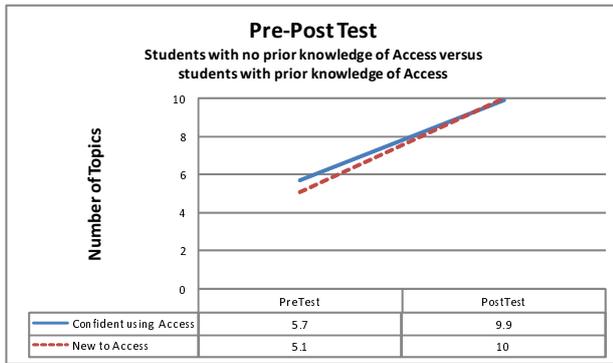


Figure 12: Students confident using Access vs students new to Access.

Regarding the second question that Woolf (2008) recommends that should be addressed, as we stated at the beginning of this section, we could not ask some of the students not to use PAT or to use a version of PAT without feedback messages or with different feedback messages. However, in Section 5.1, we have shown the students' opinion about messages received.

The distribution of number of students by the number of topics learned, is depicted in Figure 13.

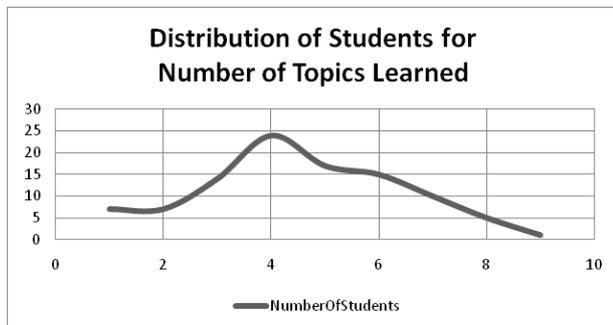


Figure 13: Distribution of students by topics learned.

The graph shows how many students learned 1 topic, 2 topics, and so on up to 9 topics - the maximum number of topics learned out of the 10 topics analysed. It can be clearly seen that the majority of them (about 70%) learned between 3 and 6 topics.

6 Conclusions and further work

The results of the evaluation showed PAT's usefulness for students' learning, as well as PAT's acceptance by both students and staff members.

These results also showed differences between students who had used Access before starting the subject, and those who had not. The differences were in the way the students used PAT and in the type of feedback they prefer. In addition, from the results of the evaluation it can be seen that the students would like to have ITSs similar to PAT for other subjects.

In addition, to show the improvement that PAT engenders in student learning, we used the data that PAT gathers as a pre-post test. The results from the test show that the students who used PAT had an average number of topics learned of 4.5 (out of the 10 most important topics analyzed), with a mode of 4 topics learned and a standard deviation of 1.87.

Because of PAT's modular structure, further enhancements can be made. The enhancements to PAT from which the students could benefit are:

- analyze (and provide help on) not just the correctness of the solution but also the readability and usability of the form or report;
- an open student model; and
- reports and statistics for teaching staff about students' learning performances.

Each of these is elaborated below.

A). When analysing the student's form or report, PAT analyses its correctness from a functional point of view. I.e. is the form or report producing the correct data? From a human user perspective though, the readability and usability of the form or report could also be analyzed.

From a readability point of view, the objects in the form should have the same size, should be aligned, and should be grouped by their meaning or function. From an usability perspective, the fields should be displayed in the most meaningful order i.e. in the same order in which the data will be entered - first name, last name and address; not last name, address, and only then the first name.

This approach would involve PAT analysing other aspects such as the relative position and size of each of the objects in the form or report.

B). An open student model would allow the students to check their profiles. This would help the students' learning by facilitating metacognitive processes and providing them with an opportunity to reflect on their progress.

C). A benefit to the teaching staff would be to generate reports and statistics about students' learning performances. Aggregated data collected by PAT could help the teaching staff identify topics that are hard to learn, suggesting areas for future improvements.

References

- Adamski, J. J. & Finnegan, K. T. (2008), *New Perspective Microsoft Office Access 2007 - Comprehensive*, Thomson Course Technology.
- Beck, J., Stern, M. & Haugsjaa, E. (1996), 'Applications of ai in education', Web Page; <http://www1.acm.org/crossroads/xrds3-1/ai.ed.html>. last visited 05 Oct 2011.
- Bhagat, S., Bhagat, L., Kavalan, J. & Sasikumar, M. (2002), Acharya: An intelligent tutoring environment for learning sql, in 'Proceedings of Vidyakash 2002 International Conference on Online learning'.
- Burns, H. L. & Parlett, J. W. (1991), The evolution of intelligent tutoring systems: Dimensions of design, in H. L. Burns, J. W. Parlett & C. L. Redfield, eds, 'Intelligent tutoring systems : evolutions in design', Lawrence Erlbaum Associates, Hillsdale, NJ, pp. 1-12.
- Card, S. K., Moran, T. P. & Newell, A. (1983), *The psychology of human-computer interaction*, L. Erlbaum Associates, Hillsdale, N.J.
- Carroll, J. M. (1990), *The Nurnberg funnel : designing minimalist instruction for practical computer skill*, MIT Press, Cambridge, Mass.
- Dollinger, R. (2010), Sql lightweight tutoring module - semantic analysis of sql queries based on xml representation and linq, in 'Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2010', AACE, Toronto, Canada, pp. 3323-3328.

- Dollinger, R., Ford, R., Helf, B. & Reimer, K. (2009), 'Ajax enabled query tool the capstone experience', *Information Systems Education Journal* **7**(49).
URL: <http://isedj.org/7/49/>
- Holt, P., Dubs, S., Jones, M. & Greer, J. (1994), The state of student modeling, in J. E. Greer & G. McCalla, eds, 'Student modelling : the key to individualized knowledge-based instruction', Vol. 125 of *NATO ASI Series F: Computer and Systems Sciences*, Springer-Verlag, pp. 3–35.
- Iqbal, A., Oppermann, R., Patel, A. & Kinshuk (1999), A classification of evaluation methods for intelligent tutoring systems, in 'Software-Ergonomie 99, Design von Informationswelten, Gemeinsame Fachtagung des German Chapter of the ACM, der Gesellschaft für Informatik (GI) und der SAP AG', Teubner, pp. 169–181.
- Knowles, M. S. (1980), *The modern practice of adult education : from pedagogy to andragogy*, Cambridge Adult Education, New York.
- Mitrovic, A. (1998), Learning sql with a computerized tutor, in 'Proceedings of the twenty-ninth SIGCSE technical symposium on Computer Science Education', ACM Press, Atlanta, Georgia, United States, pp. 307–311.
- Mitrovic, A. (2002), Normit, a web-enabled tutor for database normalization, in 'Proceedings of the International Conference on Computers in Education ICCE', Vol. 2, Auckland, New Zealand, pp. 1276–1280.
- Mitrovic, A., Suraweera, P. & Martin, B. (2004), 'Db-suite: Experiences with three intelligent, web-based database tutors', *Journal of Interactive Learning Research* **15**, 409–432.
- Mitrovic, A. & team, T. I. (2008), Constraintbased tutors, in B. P. Woolf, E. Aimeur, R. Nkambou & S. Lajoie, eds, '9th International Conference on Intelligent Tutoring Systems', number (LNCS) 5091 in 'Lecture Notes in Computer Science', Springer-Verlag, Montreal, Canada, pp. 29–32.
- Self, J. (1987), Student models: what use are they?, in P. Ercoli & R. Lewis, eds, 'Artificial Intelligence Tools in Education', Frascati, pp. 73–86.
- Suraweera, P. & Mitrovic, A. (2002), Kermit: A constraint-based tutor for database modeling, in S. A. Cerri, G. Gouarderes & F. Paraguacu, eds, 'Intelligent Tutoring Systems', Vol. 2363 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 377–387.
- Suraweera, P. & Mitrovic, A. (2004), 'An intelligent tutoring system for entity relationship modelling', *International Journal of Artificial Intelligence in Education* **14**, 375–417.
- VanLehn, K. (2006), 'The behavior of tutoring systems', *International Journal of Artificial Intelligence in Education* **16**, 227–265.
- Woolf, B. P. (2008), *Building Intelligent Interactive Tutors: Student-centered Strategies for Revolutionizing E-learning*, Morgan Kaufmann.

Using Quicksand to Improve Debugging Practice in Post-Novice Level Students

Joel Fenwick

The University of Queensland,
Earth Systems Science
Computational Centre,
joelfenwick@uq.edu.au

Peter Sutton

The University of Queensland,
School of ITEE
p.sutton@itee.uq.edu.au

Abstract

The ability to debug existing code is an important skill to develop in student programmers. However, debugging may not receive the same amount of explicit teaching attention as other material and the main expression of debugging competence is students' ability to undo problems which they themselves have injected into their assignments. Further, as the literature points out, debugging skills do not necessarily develop at the same rate as code writing skills.

This paper discusses an intervention in a second year course designed to improve students' application of simple debugging techniques. We use a puzzle based approach where students are graded based on the number of attempts they take to locate misbehaving code in a program which they did not write but whose function they understand. An existing assignment component addresses another aspect of debugging practice.

1 Introduction

The context for this work is a second year course in systems programming (networks and operating systems). Because of its place in the degree program, it also does triple duty as a means to force students to improve their programming skills and to learn the language used in the course (C). All students enrolling in the course have some exposure to C but much of their basic training has been in Python or Java.

This setting is a little different from the typical setting in the literature (McCauley et al. 2008, Fitzgerald et al. 2008), in that we are not (or should not be) dealing with absolute novices any more. These are students who have some level of programming skill even if they do not have much initial familiarity with C. However, "debugging is a skill which does not immediately follow from the ability to write code." (Kessler & Anderson 1986)[p208]. Following on from an earlier working group, an ITiCSE 2004 group (Lister et al. 2004) considered whether deficiencies in programming ability *after* initial programming courses were due to lack of problem solving skill or were in fact due to a fragile knowledge of programming and code reading ability. However, "reports on interventions designed to improve students' debugging skills have not been common in recent literature." (McCauley et al. 2008)[p83]

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the 14th Australasian Computing Education Conference (ACE 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 123, Michael de Raadt and Angela Carbone, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

The course has four assignments. The first, third and fourth assignments are traditional programming assignments where the students must write whole programs (possibly making use of a provided solution for the previous assignment). The focus of the second assignment is debugging. In 2011, it consisted of two components: the binary bomb and quicksand (both explained below). Both parts were conducted electronically and only required a network connection to a school server.

The binary bomb is a modified version of an assignment run at Carnegie Mellon University (Bryant & O'Hallaron 2001). This component tests students' ability to use a debugger. Students are given a pre-compiled program with some of the debug symbols removed as well as a small part of the source code. They must use the debugger to examine the workings of the program to determine the passwords to "defuse" the bomb. We have used the binary bomb for a number of years and it seems popular with the students with the puzzle solving aspect mentioned in particular.

However, debuggers are only as useful as the questions they are asked and some students start to view the debugger as the first port of call in solving any problem even when they do not know what they are looking for. Further, some students would reach the end of the course and still start their requests for help with "My program doesn't work." That is, they did not seem to be able to locate or describe the problem more precisely. There is also a school of thought [typified by Linus Torvalds' refusal to incorporate kernel debuggers into the Linux Kernel (Torvalds 2000)], that over reliance on a debugger produces sloppy programmers. James et al. note that "the more tools offered, the less students think for themselves. They try to get the tool to do the thinking..." (James et al. 2008)[p28]

Addressing these and other problems is the purpose of the quicksand component, and the focus of this paper. However, since this is not an introductory course, some of the students do already have good debugging technique and care must be taken not to bore these students while trying to lift the others.

1.1 Debugging Challenges in C

Debugging programs written in C presents some challenges. These are by no means unique to C but when they happen in C (and similar languages) they tend to be more spectacular and less help is available. Possible problems include:

1. Compilers (more specifically optimisers) can change things and sometimes they get it wrong.
2. Functions can have bugs or undocumented behaviour.

3. A badly written statement can create problems which only appear much later in seemingly unrelated places (e.g. heap corruption).
4. The lack of structured error handling.
5. Corruption from non-thread safe actions.¹

Regarding the first point, (some) students seemed to believe quite strongly that the compiler was transparent. The idea that what was executing could be different to what they saw when they looked at the source was problematic. One of the authors' first experiences with this as a student was realising that an optimiser had decided to run a loop in reverse.

This is not to say that trusting the compiler or documentation is a bad starting heuristic, but it seemed to be a weakness in the students' understanding that translation failure was unthinkable. It may be that even just allowing for that possibility might enable students to perform the necessary testing to discover the real problem.

2 Quicksand

The quicksand component (created by us) is intended to develop and test a complementary set of skills to that of the binary bomb. Instead of using the debugger to step through and examine variables, the students can make small modifications to a piece of source code (typically printing the values of variables), then request that the code be recompiled and run against a set of tests. The students can use output from this run to locate bugs in the program.

The students were told quite explicitly that they were looking for the original cause of the bug not just the location where symptoms became apparent. For example an uninitialised variable might not cause any obvious effect until much later.

As in the work by Fitzgerald et al. (Fitzgerald et al. 2008), we only require students to identify the line on which problems occur and not to actually fix the problem. Firstly, as they note (citing Kessler and Anderson), "the skills required to understand the system are not necessarily connected to the skills required to locate the error." [p95] Secondly, it may not be possible to theorise about the cause of a problem until its location has been constrained. Much time has been wasted looking for a problem in the wrong place because programmers jump to conclusions. Finally, since the students are supplied with code to debug rather than writing it themselves, we need to address the issues of understanding and context (Fitzgerald et al. 2008). To this end, the supplied code is a (suitably buggy) sample solution to Assignment 1. This means that the students will be familiar with the purpose of the program and what it should be doing. It is also sufficiently large that the entire program cannot be held in mind at one time and some investigation is required to locate bugs precisely.

Lines which may contain bugs have a "tag" at the end in the form of a right-justified comment (e.g. `/* QS:f8y5d */`). We use tags rather than line numbers because line numbers will vary when students add new lines to the source. Also tags are harder to mistype. Lines which are not possible bug locations do not receive a tag but are marked `/* QS: */`. This makes it easy to distinguish lines which are in the supplied source as opposed to lines added by students.

When editing the source, students may insert additional lines but may not modify any existing lines.

¹This is another thing which can cause seemingly inexplicable behaviour — we do not address thread safety issues in this exercise.

This is to discourage students from trying to reimplement sections they are suspicious of. In practice such an approach would only work in a limited number of instances.

We also limit the number of extra lines which the students can have in the source at any one time. This is to discourage the creation of massive quantities of debug information which is then impossible to follow. Instead, we want students to focus their attention and refine their theories as to the location of the problem. If they exceed this limit, they must remove some lines or get a clean copy of the source before they will be able to recompile.

2.1 The quicksand tool

The quicksand tool itself has four commands:

- **get** — puts a "clean" copy of the source to be debugged in the current directory.
- **test** — processes and compiles the student's modified source and runs the set of tests. The outputs from the tests are placed in the student's directory. At no time does the student have access to the compiled binary itself.
- **guess tag** — Records the student's "guess" that the tagged line contains a bug. It will tell the student if their guess is correct.
- **status** — Reports how many attempts the student required to locate each bug as well as their current and maximum possible marks.

To encourage students to use online documentation, the assignment specification and tool instructions were only available as a man page.

2.2 Marking

The marks gained for correctly determining the location of a bug were

$$\frac{T}{B} \cdot 0.9^{g-1}$$

where T is the total marks for this part, B is the number of bugs in the system and g is the number of guesses since the previous correct guess. This follows a similar "exponential decay" scheme used in the bomb and ensures that more wrong answers will decrease the mark but eventually answering correctly still gives more marks than giving up. Since the number of tags in the program is limited though, we did impose a limit of 40 on the total number of guesses for all bugs. Only eleven of approximately 140 students used all 40 guesses.

Note that editing and testing are free actions, the students can do as much testing as they wish without it influencing their marks.

2.3 Bugs

All students were given the same piece of code as a starting point but different combinations of bugs were introduced for each student. The bugs were chosen so that any one of the bugs would cause at least one of the tests to fail. That is, the students could use test failures as the starting point to trace the bug.

Bugs introduced by quicksand fall into two broad categories:

- Visible in source — the source given to the student has a logic error in it. The student could find it by inspection. Examples include:
 - Uninitialised or incorrectly initialised variables.
 - incorrect loop limits or steps
 - inverted if conditionals
 - invalid memory access.
- Hidden — These can not be found by inspection since they do not appear in the source which the students are given. When the student runs quicksand’s test command, their source is transformed. Some lines are replaced or modified before the program is finally compiled. For example:
 - Statements can be skipped (removed from the source).
 - Assignments and initialisations changed to assign different values.
 - Loops can finish early or skip iterations (modified limits or step)
 - Variables can be modified unexpectedly.

It is important to understand that while the precise details and causes of these “hidden bugs” are artificial, the symptoms are not unreal. Although some types (e.g. statements being skipped) are thankfully rare² “in the wild”.

The “hidden” bugs are representative of instances where either the mental model of the programmer (expressed in code) does not match what is actually there; or where the documentation is incorrect or (in rare cases) where the compiler or standard libraries have bugs.

The possibility of bugs in code not written by the students represents a point of difference between beginner and later programmers. Courses for beginners will focus on a relatively small, well tested and well understood subset of the standard libraries for their language. Later on however, programmers need to be able to make a distinction between source and running code; between what documentation or their understanding suggests and what is actually there.

Some work (Lee & Wu 1999, Ahmadzadeh et al. 2007) used debugging exercises as a means to improve general programming skill. There is nothing wrong with this approach and training programmers to write less bugs is a good goal. However, debugging is not solely a means to rectify one’s own coding faults, which could be avoided by writing more carefully. Even a perfect programmer needs to be able to debug. Debugging is also an independent skill which may be required whenever code is brought together or when some part of the environment changes.

So how do the students find bugs that they can’t see? Actually, whether the bugs are immediately visible is not immediately relevant to finding them. At this stage in their development, students are writing programs which are too large to be completely comprehended at one time. Eisenstadt’s “war stories” article (Eisenstadt 1997) contains a number of memorable terms for difficulties in finding bugs: the “Cause/Effect Chasm” where the cause and effect were too far apart to be easily found; and WYSIPIG — “what you see is probably illusory guv’nor” where

²One of the authors has encountered such errors a number of times including, in an unrelated piece of code while developing quicksand.

the programmer misreads or misunderstands what they are looking at. In a large system without knowledge of the (general) location of the problem, finding problems by inspection is not feasible. So initially, visibility is not critical. Instead, all the bugs introduced by quicksand can be located using two generally applicable techniques.

- Strategically placed output statements to trace execution flow. Once the symptom has been identified (e.g. a crash or incorrect output).
- “binary search” — output the values of critical expressions at and before the symptom location, choose a point between them and repeat until the cause of the symptom is found.

These are pretty rudimentary methods but they are useful and (some) students do not seem to be applying them. But why not use more advanced tools for this exercise? After all Lieberman wrote that “It is a sad commentary on the state of the art that many programmers identify ‘inserting print statements’ as their debugging technique of choice.” (Lieberman 1997)[p27] Firstly, we want to encourage students to use hypothesis testing rather than trial and error (Ahmadzadeh et al. 2007). This constrained environment prevents them from relying on other tools too much. Secondly, James et al. (James et al. 2008) suggest that more advanced tools may actually hinder students from learning good technique. Thirdly, there are a number of common environments (such as web programming) where more advanced tools may not be available but simple techniques work everywhere (James et al. 2008).

In part, this is an application (although not a rigorous one) of the *malicious adversary* concept from theoretical computer science. That is, if you can find bugs using these techniques when you are being deliberately sabotaged, then they will be useful under normal conditions as well.

3 Security and Integrity

This exercise combines two factors which make the security of the system a concern. Firstly, quicksand needs sufficient privileges to access the database and record attempts. Secondly, students will be able to inject “arbitrary” code into the test program.

With this in mind we tried to impose limits to make the code they could inject less arbitrary. Lines added by students could not contain #³, any of a number of system functions nor any raw assembly or system calls.

Privilege issues were dealt with by dropping to student privileges whenever interacting with student code and by preventing students from attaching a debugger to any of the programs involved. Further technical details are beyond the scope of this paper.

There are other issues related to the integrity of the system that don’t relate to security. It is not enough that the students work out the answers, we want them to use the correct technique in doing so. An example of an incorrect technique is removing “suspicious” lines and by means of comments, loop bodies which never execute and so on. Our intent is that the students insert small “probes” to determine what is going on, not switch out chunks of code. For this reason the *real* source (as opposed to the version the students edit) contains extra calls to check that statements are not being executed out of order. Further, some bugs add additional calls to enact their

³or any of its equivalents

bad behaviour. If the student inserts an incomplete line immediately before such a call, then the compiler may “helpfully” tell the student that the next line contains `garbleData()`. To avoid this, only the line number of compile errors are reported back to students.

While a number of students noticed the forbidden words and symbols list, only one student (that we know of) accidentally ran into the anti-reordering checks.

A non-technical aspect of assessment integrity is the possibility of collusion. What if the students discuss their bugs? Does this disadvantage students who start work early? While sharing answers is possible⁴, it presents a number of difficulties for students. First, students would need to know that they had been assigned the same bug. Even if they establish that they have some bugs in common, this does not allow them to infer that any of their other bugs are the same. Establishing with certainty that you share a bug with someone else (without using an attempt) requires the same skills as finding the bug “the proper way.” Now, other types of collusion are possible, such as one student doing much of the work for another student but this risk is (we believe) no higher than a traditional assignment.

Secondly, sharing answers with students who have not put the work in acts against the students’ self-interest. Suppose Student A took 3 attempts to identify the location of a bug, and they establish that Student B also has this bug (not withstanding the difficulties in doing so). Now suppose that Student A gives the answer to Student B who has made less than 3 attempts, Student A has given Student B an advantage, B will now get more marks for that bug than A.

Thirdly, students who start work early are perhaps less likely to cheat.

4 Results and Reflection

The biggest teaching challenge here is to get the students to not blindly trust their intuition but rather to test the safety of their ground before relying on it too heavily⁵. At the same time, we need to show them that reason and logic still apply in debugging situations.

Fitzgerald et al. describe students in their study as having a “stubborn desire to understand and debug code through reading alone.” (Fitzgerald et al. 2008) This was borne out in this work where students took the source, worked on it and tested it entirely outside of quicksand. This created confusion when the code ran differently under the normal compiler as opposed to the malicious quicksand. In future this would need to be prevented. The easiest approach would be to ensure that the students are not given the source for all routines and hence would not be able to compile independently.

We have a number of sources of information to use in evaluating quicksand.

1. Anonymous surveys taken immediately after the assignment.
2. Logs of the types of questions asked in tutorials.
3. Questions asked on the course online discussion group.
4. Teacher impressions.

⁴We have no evidence that this occurred.

⁵Hence “quicksand”.

We also have university end-of-course surveys but the responses did not reveal anything about this assignment.

Since the survey was voluntary and students were not required to attend tutorials or post to the discussion group, we need to be careful about what conclusions we draw. A student who has no problems that they can not fix for themselves will not show up. Also, considering only those students who ask for help in tutorials tends to give a worst case approximation of how things are going. A lack of more sophisticated questions could either mean that people are better able to fix their own problems and do not need to ask or that they are getting stuck at a basic level. Taken together, this means that we may only be able to consider a lower bound on improvements.

We will now discuss each information source in more detail.

4.1 Assignment surveys

This survey asked the students a number of questions about quicksand and the binary bomb. Some were to be answered on a five point scale while others were free text. Of the 77 respondents, 49% said that they had learned a lot from quicksand, while 57% said that quicksand had improved their confidence in their debugging abilities.

Interestingly 12% of students said that they already knew and used such techniques (and as such are unlikely to report learning a lot) and some of them still reported improved confidence.

The importance of retaining the binary bomb component is shown by the fact that 51% of respondents said that they did not know how to use the debugger prior to this course⁶.

Taking both parts of the assignment (quicksand and binary bomb) together, 60% said that they had learned techniques that they could have used in the previous programming assignment. 52% said that they were more systematic when debugging now.

The free text responses indicate that some students were not convinced that programs could go wrong in the way that our exercises did. As such a wider set of possible problems is probably indicated. For example, a bigger focus on function calls misbehaving (where they can not see the source) would probably be more acceptable to them. A number of students seemed to believe that using `printf` wasn’t real debugging. This seems to put even simple techniques in the category of fragile knowledge for some students. Students did not seem to be surprised or confused when these techniques are pointed out, but did not seem to have considered applying them to solve their problems so we are dealing specifically with “inert knowledge” (Perkins & Martin 1986).

4.2 Tutorial Logs

Each tutor logged the number of questions they answered by category. Questions about debugging were classified into the following categories. They are ordered by the sophistication of the question (roughly how much work the student has put in before they ask the question):

- debugA — Questions of the form “It doesn’t work.” or “I’m failing test #5.”⁷. Here there

⁶It is not clear from these answers whether all of those students are referring to all symbolic debuggers or just non-IDE ones.

⁷Students were given a set of automated tests which they could use to test their assignments against some parts of the spec.

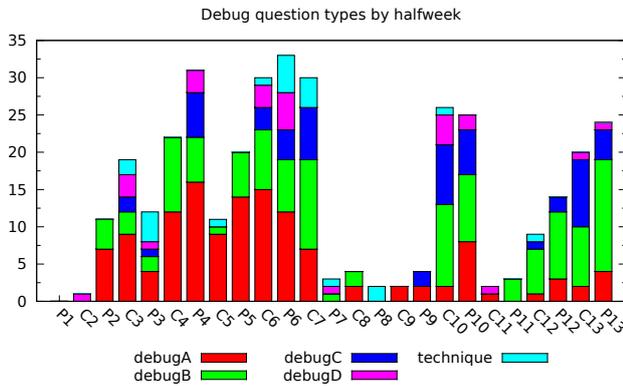


Figure 1: C_n denotes tutorials conducted in the first half of the week while P_n denotes tutorials from the second half.

is no description of the problem and no work apparent towards identifying the location, cause or triggers for the problem.

- debugB — Some basic effort has been made to locate the problem. Some ability to describe the specifics of the problem.
- debugC — The student demonstrated good technique.
- debugD — The student did all the right things but were prevented from finding the problem due to missing some knowledge or a misunderstanding. Essentially, their technique was not the problem.
- technique — the student was not asking for help fixing a particular problem but wanted to know about debugging techniques in general.

Tutorials in this course were run in two half-weeks (Monday to Wednesday morning and Wednesday afternoon to Friday afternoon). Students were required to enrol in one session in each half-week. However, tutorials were not compulsory and while some weeks were set aside for the tutors to teach extra material, the majority were general help sessions. As such, the attendance varied significantly depending on the time to deadline.

The breakdown can be seen in Figure 4.2. The assignments were due at the end of Weeks 4, 7, 10 & 13 respectively. Note that once the questions start to increase again for Assignment 3, the more sophisticated categories dominate the debugA questions.

4.3 Discussion Posts

The course had a very active online discussion group. We separated out the threads which asked for help or suggestions for fixing bugs in Assignments 1, 3, 4. These were classified according to the initial question using the same scheme used for tutorial questions. The number of questions (and the number of different students asking them) increased from Assignment 1 to Assignment 3 (the assignment the students seemed to find most difficult) and then dropped below the level of Assignment 1 for Assignment 4. The majority of the questions fit into debugA — either little description of the problem or a request for suggestions about where to get started. In Assignment 4,

there was an increase in the proportion of questions which described features of the system the students were trying to fix rather than a particular test failure.

4.4 Teacher Impressions

As well as visiting tutorials from time to time one of the authors conducted intensive help sessions just before the deadline for Assignments 3 and 4. Unfortunately the question types from these sessions were not logged. The first session was not particularly well attended. The second session saw at least 40 students and from memory, the majority of questions were debugB or above.

5 Implementation Considerations

What is required in order to run an assignment like this? In terms of software infrastructure, A database for storing marks and infrastructure (quicksand in our case) for distributing source and compiling student modified versions will be needed. The following should also be considered.

- *The security of the marks record.*
Students must not be able to coerce the system into modifying marks for themselves or other students. They must not be able to view the marks of other students.
- *Backdoor solutions.*
What mechanisms are available for bringing other code into the test program? For example, in Java, it would not be sufficient to block `import` since classes could be pulled in via their full name or using Java's reflection API⁸. In Python there is `eval()`, the `pdb` debugger and probably others. This is not to suggest that these languages can not be secured⁹ but the implications of these features need to be considered.
Can your hidden modifications be exposed in compile errors or exception traces?
- *Types of Bugs to inject.*
Firstly, the bugs must be plausible in your source language¹⁰. For example a string changing for no apparent reason in Python or Java (where String objects are immutable) would be a bad choice. Secondly, bugs that do not require modification of the student's submitted source require much less machinery and are less fragile than things like skipping particular statements. Limiting oneself to function calls which misbehave under certain conditions would save a lot of work.

6 Conclusions

From the survey, more than half the students reported increased confidence in their debugging abilities as a result of quicksand. This and more than half reporting they were more systematic in their debugging from the assignment as a whole, are encouraging. Looking at the online discussion is less positive but students may have been less willing to post detailed questions. Either because they weren't sure how to express them in text or because of the warnings they were given about posting code. Overall, for non-pathological bugs this approach shows promise.

⁸API = Application Programming Interface

⁹A reviewer suggests `SecurityManager` in the case of Java.

¹⁰That is, the language of the program the students are editing

The *possibility* that the compiler or libraries misbehave should be part of the students' thinking. It appears though that more explanation needs to be given, perhaps with real world examples, for students to accept this.

Acknowledgements

This work was supported in part under AuScope sustainability funding.

References

- Ahmadzadeh, M., Elliman, D. & Higgins, C. (2007), 'The impact of improving debugging skill on programming ability', *ITALICS* **6**(4), 72–87.
- Bryant, R. E. & O'Hallaron, D. R. (2001), 'Introducing computer systems from a programmer's perspective', *SIGCSE Bull.* **33**, 90–94.
URL: <http://doi.acm.org/10.1145/366413.364549>
- Eisenstadt, M. (1997), 'My hairiest bug war stories', *Commun. ACM* **40**, 30–37.
URL: <http://doi.acm.org/10.1145/248448.248456>
- Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L. & Zander, C. (2008), 'Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers', *Computer Science Education* **2**(18), 93–116.
- James, S., Bidgoli, M. & Hansen, J. (2008), 'Why Sally and Joey can't debug: next generation tools and the perils they pose', *Journal of Computing Sciences in Colleges* **24**, 27–35.
URL: <http://portal.acm.org/citation.cfm?id=1409763.1409770>
- Kessler, C. M. & Anderson, J. R. (1986), A model of novice debugging in lisp, in 'Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers', Ablex Publishing Corp., Norwood, NJ, USA, pp. 198–212.
URL: <http://act-r.psy.cmu.edu/publications/pubinfo.php?id=220>
- Lee, G. C. & Wu, J. C. (1999), 'Debug it: A debugging practicing system', *Computers & Education* **32**(2), 165 – 179.
URL: <http://www.sciencedirect.com/science/article/pii/S0360131598000633>
- Lieberman, H. (1997), 'Introduction to the special issue on the debugging scandal', *Commun. ACM* **40**, 26–29.
URL: <http://doi.acm.org/10.1145/248448.248455>
- 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 Bull.* **36**, 119–150.
URL: <http://doi.acm.org/10.1145/1041624.1041673>
- McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simone, B., Thomas, L. & Zander, C. (2008), 'Debugging: a review of the literature from an educational perspective', *Computer Science Education* **2**(18).
- Perkins, D. & Martin, F. (1986), Fragile knowledge and neglected strategies in novice programmers, in S. E. & I. S., eds, 'Empirical Studies of Programmers', Norwood, NJ: Ablex Publishing Co., pp. 213–229.
- Torvalds, L. (2000), 'Re: Availability of kdb', Linux Kernel Mailing List.
URL: <http://lkml.org/lkml/2000/9/6/65>

Coming to terms with Bloom: an online tutorial for teachers of programming fundamentals

Richard Gluga

School of Information Technologies
University of Sydney
Sydney NSW 2006 Australia
rgluga@it.usyd.edu.au

Judy Kay

School of Information Technologies
University of Sydney
Sydney NSW 2006 Australia
judy.kay@sydney.edu.au

Raymond Lister

Faculty of Information Technology
University of Technology
Sydney NSW 2006 Australia
raymond.lister@uts.edu.au

Sabina Kleitman

School of Psychology
University of Sydney
Sydney NSW 2006 Australia
sabinak@psych.usyd.edu.au

Tim Lever

Faculty of Engineering and Information Technologies
University of Sydney
Sydney NSW 2006 Australia
tim.lever@sydney.edu.au

Abstract

This paper describes a web-based interactive tutorial that enables computer science tutors and lecturers to practice applying the Bloom Taxonomy in classifying programming exam questions. The structure, design and content of the tutorial are described in detail. The results of an evaluation with ten participants highlight important problem areas in the application of Bloom to programming assessments. The key contributions are the content and design of this tutorial and the insights derived from its evaluation. These are important results in continued work on methods of measuring learning progression in programming fundamentals.

Keywords: programming, Bloom, maturity, competence, learning progression, assessment

1 Introduction

A typical university Computer Science Bachelor degree is three to five years long (in the case of combined/double degrees). The degree consists of a series of semester-long subjects. The design of the curriculum for a degree must enable students to steadily progress in acquiring discipline skills, as reflected in the ACM/IEEE CS Curriculum (2008). So, for example the first programming subject in this sequence, (CS1) may assume no computing pre-knowledge; students typically learn basic programming fundamentals (e.g., variables, loops, control structures, functions, syntax). A later programming subject in the sequence, such as Data Structures, typically assumes students have basic competence in using these programming fundamentals. This Data Structures subject would enable students to learn new concepts (e.g., lists, maps, sets, sorting algorithms). At the same time, students should increase their level of competence on the programming fundamentals introduced in the first programming subject.

This example highlights the progressive nature of skill development and maturity in a Computer Science degree. Students are not expected to immediately master all new concepts in a subject. Rather, their competence level should increase as they progress from one subject to the next. In order to support this progressive learning model, learning activities, assessment tasks and exams need to be appropriately structured to teach and assess students at the appropriate level of competence. That is, a final exam in the first programming fundamentals subject should be designed to assess whether students have reached the competence level that is appropriate for that stage. This design should also account for the range of learning achievements across the class. Any student who earns a passing grade should have basic levels of competence. The top-performing students should be able to demonstrate a more advanced level of competence. The second subject (CS2) should then have assessed learning activities and exams that require a more advanced level of competence of programming fundamentals and a more basic level of competence of the new concepts.

The issue that arises out of this discussion is how to classify teaching activities and assessments at a particular level of competence. That is, how can a lecturer write an exam question that assesses a programming fundamentals concept at a novice level vs. a more advanced level? Additionally, given an existing exam paper, how can a lecturer judge what level of competence is required to correctly answer a particular exam question?

Several theories exist that can be used in this classification, the most prominent of which are Bloom's Taxonomy (Bloom 1956), the SOLO Taxonomy (Biggs and Collis 1982, Sheard et. al. 2008) and Neo-Piagetian development theory (Morra et. al. 2007, Lister 2011). In this paper we focus on Bloom, as it is currently used in the ACM/IEEE CS Curriculum (2008) and will be used in the revised 2013 Curriculum (ACM/IEEE 2013). Bloom's taxonomy is also commonly recommended by university teaching support services as a guide for learning outcome specification (e.g., Centre for Learning and Professional Development, University of Adelaide, 2011; Learning & Teaching Centre, Macquarie University, 2008; Centre for the Advancement of Teaching and Learning, University of Western Australia, 2005). Bloom offers the potential for more principled design of the curriculum as it helps

classify the level of learning to be achieved in each subject and the curriculum design can ensure that there is progression. Additionally, Bloom should help a lecturer design examination questions so that they assess learning at the appropriate level of competence.

To do this though, a lecturer requires an understanding of the Bloom Taxonomy, how it applies in a computer science context, and how it can be used to classify programming assessment tasks. Our motivation is to enable lecturers to gain this understanding of Bloom so that they can apply it to their own exam questions. This will enable computer science lecturers to review existing exam papers and design future exam papers with a greater awareness of progression and competency levels, thus avoiding either overly-high or under-ambitious expectations.

To this end, our contribution is a computer science contextualized web-based tutorial on the Bloom Taxonomy with interactive examples, user self-explanation and self-reflection. The tutorial is a useful resource in training computer science educators on the application of Bloom in classifying programming assessment questions. The results from the evaluation of this tutorial are useful in identifying where Bloom is used inconsistently due to different assumptions about the learner, different interpretations of the Bloom categories, or a misunderstanding of the categories. The tutorial and Bloom insights are important inputs to future work on measuring learner progression in computer science and future revisions of the ACM/IEEE Computer Science Curriculum.

2 Background

Benjamin Bloom himself once said that the original Bloom Handbook (Bloom 1956) was “one of the most widely cited yet least read books in American education” (Anderson 1994). The taxonomy is a behavioral classification system of educational objectives. The framework specifies six categories, namely, Knowledge, Comprehension, Application, Analysis, Synthesis and Evaluation. Knowledge is the simplest behavior, with each category thereafter being more sophisticated. That is, a Knowledge level objective or assessment task requires a student to simply recall information from memory. In contrast, a Synthesis level task requires students to apply what they have learnt to create new and unique works. A very brief description of the categories follows (adapted from Anderson 1994):

- *Knowledge* – recalling of information
- *Comprehension* – interpreting, translating or reordering of concepts, applying a given abstraction
- *Application* – identifying an appropriate abstraction to solve a problem without being prompted
- *Analysis* – breaking down a problem or communication into parts and identifying the relationships between the parts
- *Synthesis* – Identifying and putting together abstractions to create a new and unique artifact or solution to a non-trivial problem

- *Evaluation* – Commenting on the validity of a work with respect to implicit or explicit criteria

Lister (2001) discussed the problem of first year subjects being overly-ambitious and requiring students to manifest competence at the Bloom Synthesis or Evaluation levels. This is an unrealistic expectation if students have not had the chance to steadily progress through the first four levels of mastery. The consequence of this is that first year CS subjects specify learning objectives that require a high-level of competence, but consequently have lax assessment marking schemes that allow students to pass under false pretences with very little competence in the specified objectives. Lister also argued that the sequencing of material should be based explicitly on a model of learning progression, and proposed Bloom’s Taxonomy (Bloom 1956) as a potential model.

Further, Lister (2001) suggested CS1 students should be expected to primarily operate at the first two Bloom levels (Knowledge and Comprehension). “CS1 cannot produce accomplished programmers. That is the task of an entire sequence of programming subjects.” Students should thus not be expected to operate at the higher Bloom levels (Synthesis/Evaluation) by writing original code in a first semester subject, yet this appears to be a common occurrence.

This trend towards overly ambitious first year subjects was also noted by Oliver and colleagues (2004). Here the authors took six subjects from a single Australian IT degree, and invited four lecturers to categorize the assessment questions on the original Bloom Taxonomy scale. The authors used the weighting of each question and the Bloom categorization to calculate a Bloom Rating for each subject as a whole. The results showed a first year, first semester programming subject had a weighted Bloom Rating of 3.9, i.e., somewhere between Application and Analysis. The first year, second semester subject had a rating of 4.5. Students in this stream were thus expected to rapidly progress through the lower levels of the scale.

Oliver and colleagues (2004) also highlighted the inconsistencies in applying Bloom to computer science exam questions. For the one example question presented in the paper, the four participating lecturers identified four distinct Bloom classifications, ranging from Knowledge to Analysis. Whalley and colleagues (2006) found the use of Bloom’s taxonomy for rating the cognitive complexity of programming MCQ’s “challenging even to an experienced group of programming educators.” The difficulty was attributed to either some deficiencies in Bloom, or “the authors current level of understanding of how to apply the taxonomy.”

The ACM/IEEE CS Curriculum (2008) supports the notion of gradual student progression, although it does not give direction as to how this progression should be implemented. The curriculum specifies a collection of learning objectives and topics, organized by knowledge area. The learning objectives are based on the revised Bloom Taxonomy (Anderson et al, 2001). As an example, the Programming Fundamentals / Data Structures knowledge area specifies the following nine learning

objectives (the italicized verbs are indicative of the Bloom levels):

- *Describe* the representation of numeric and character data.
- *Understand* how precision and round-off can affect numeric calculations.
- *Discuss* the use of primitive data types and built-in data structures.
- *Describe* common applications for each data structure in the topic list.
- *Implement* the user-defined data structures in a high-level language.
- *Compare* alternative implementations of data structures with respect to performance. *Write* programs that use each of the following data structures: arrays, strings, linked lists, stacks, queues, and hash tables.
- *Compare* and contrast the costs and benefits of dynamic and static data structure implementations.
- *Choose* the appropriate data structure for modeling a given problem.

These objectives show a spread of competence levels ranging from Bloom Knowledge (*describe*) to Bloom Synthesis and Evaluation (*write, implement, compare & contrast*). The 2013 curriculum is expected to continue along similar lines, but will likely use a simplified Bloom Taxonomy consisting of only three categories to identify the depth of understanding: Knowledge, Application and Evaluation (IEEE/ACM CS 2013).

Thompson and colleagues (2008) attempted to contextualize the revised Bloom Taxonomy to computer science. They ran an experiment where five educators were asked to analyze six first-year computer science final exam papers and categorize each question on the Bloom scale. The results showed significant disagreement between the rankings performed by different educators. This was attributed to some having implicit knowledge of how the subject was taught, and hence had a better understanding of the cognitive processes of the students undertaking the exam papers.

Thompson and colleagues (2008) however did not discuss the educators' prior knowledge of the Bloom Taxonomy, or its application in a computer science context. It was only after the educators had a chance to collaborate and discuss each classification that they reached consensus on each exam question.

3 Programming Bloom Tutorial

3.1 Introduction

ProGoSs (Program Goal Progression) is a research project on curriculum mapping and learning progression in university degree programs. It is an online web-based system that allows university educators to tightly link the teaching activities and assessments in each subject to important curriculum learning objectives, such as those specified in the ACM/IEEE CS Curriculum (2008). This enables educators to optimize the sequence of subjects, topics and assessments, so as to provide maximum curriculum coverage. It also aims to distinguish between

the bare-passing student and the top-performing student in terms of the learning objectives covered and the level of competence for each.

In this particular experiment, we are evaluating the use of the Bloom Taxonomy for classifying computer science assessment/exam questions.

The experiment requires participants (computer science educators) to complete our interactive Bloom Taxonomy tutorial. The intention of this tutorial is to bring participants up to speed on how to apply Bloom in introductory programming. Participants are asked to read about the Bloom Taxonomy and practice some classification examples. Participants are asked for self-explanations and self-reflections on their understanding during the tutorial. The experiment takes less than sixty minutes to complete.

3.2 Pre-Survey

The tutorial commences with a short pre-survey containing three questions, as shown in Figure 1.

Figure 1: Pre-Survey

After completing and submitting the pre-survey, participants are presented with an introduction to Bloom, which is reproduced in the following section.

3.3 Bloom Introduction

The Bloom Taxonomy is a framework for classifying learning objectives into different categories of cognitive behaviors. It describes six categories as follows (from the most sophisticated to the least sophisticated):

6. Evaluation	Most Sophisticated
5. Synthesis	
4. Analysis	Moderately Sophisticated
3. Application	
2. Comprehension	Least Sophisticated
1. Knowledge	

Figure 2: Tutorial Bloom Categories

Each category builds upon the cognitive behaviours found in the preceding categories. The six levels form three pairs where the lower element of a pair focuses upon providing an artifact and the second element in the pair is used to demonstrate understanding of such an artifact. Students are typically expected to climb up through the Bloom Taxonomy as their mastery of a discipline deepens. That is, novices would start at the knowledge

level, whereas seniors would be expected to perform at the higher levels.

3.4 Category Descriptions

Participants were required to click on each tab (as seen in Figure 3) to read the category description and self-rate their understanding for each category before proceeding to the next tab. A progress bar filled up as they completed each tab. A note informed participants that they were not expected to become an expert in Bloom just by reading these descriptions, but hopefully they should improve their understanding as they progressed through the tutorial.

1. Knowledge 2. Comprehension 3. Application 4. Analysis 5. Synthesis 6. Evaluation

Knowledge emphasizes the **recall of information**. The recall situation is very similar to the original learning situation. The knowledge category differs from the others in that **remembering is the major psychological process** involved, while in the others the remembering is only one part of a much more complex process of relating, judging and reorganizing.

Key Verbs: know, define, memorise, repeat, recall, record, list, name, relate, review, tell

Example Exam Question: What is the return data type of the following function?

```

1 public int getMin(int[] data) {
2     int min = 0;
3     for(int i = 0; i < data.length; i++) {
4         if(data[i] < min) {
5             min = data[i];
6         }
7     }
8     return min;
9 }

```

Explanation: This is a knowledge level question as the student needs to recall what a return type is to answer correctly.

Please answer the following question before proceeding to the next tab.

Based on your understanding of this category, how confident are you in being able to correctly identify a programming exam question at this Bloom level?

6%

Figure 3: Category Description

The categories are described and exemplified within the tutorial system as follows.

3.4.1 Knowledge

Knowledge emphasizes the *recall of information*. The recall situation is very similar to the original learning situation. The knowledge category differs from the others in that *remembering is the major psychological process* involved, while in the others the remembering is only one part of a much more complex process of relating, judging and reorganizing.

Key Verbs: know, define, memorise, repeat, recall, record, list, name, relate, review, tell.

Example: What is the return data type of the following function?

```

public int getMin(int[] data) {
    int min = 0;
    for(int i = 0; i < data.length; i++) {
        if(data[i] < min) {
            min = data[i];
        }
    }
    return min;
}

```

Explanation: This is a knowledge level question as the student needs to recall what a return type is to answer correctly.

3.4.2 Comprehension

Comprehension requires an *understanding of the literal message* contained in a communication. In reaching such

understanding, the *student may change the communication in his mind* to some parallel form more meaningful to him. That is, *the student may translate the communication* into another language or other terms. The student may also interpret the individual parts of a communication and re-order the ideas into a structure more meaningful to him.

Comprehension differs from Application in that the thinking is based on what is explicitly given, rather than on some abstraction the student brings from other experiences to the situation.

Key Verbs: restate, discuss, describe, recognise, explain, express, identify, locate, report, operate, schedule, shop, sketch.

Example: What is the return value of the function below when called with the following input data array: {3,7,2,9,4}.

```

public int getMin(int[] data) {
    int min = 0;
    for(int i = 0; i < data.length; i++) {
        if(data[i] < min) {
            min = data[i];
        }
    }
    return min;
}

```

Explanation: This is a comprehension level question as the student is required to understand and trace through the code to derive at the correct answer.

3.4.3 Application

Application requires the student to apply an appropriate abstraction *without having to be prompted* as to which abstraction is correct or *without having to be shown how to use it in that situation*. In a comprehension problem the student would be specifically told which abstraction he should use.

Key Verbs: translate, interpret, apply, employ, use, demonstrate, dramatise, practice, illustrate, criticise, diagram, inspect, debate, inventory, question, relate, solve, examine.

Example: Write a function to return the minimum value from an integer array that is passed as a parameter.

Explanation: This is an application level question as the student is required to write the code using an abstraction that's not already there (e.g., a loop, an if statement, a local variable).

3.4.4 Analysis

Analysis emphasizes the *breakdown of the material into its constituent parts* and *detection of the relationships of the parts* and of the way they are organized to form the whole. In comprehension, the emphasis is on the grasp of the meaning and intent of the material. In application it is on remembering and bringing to bear upon given material the appropriate generalizations or principles.

Analysis is the ability to identify unstated and/or incorrect assumptions in code, to recognize software design patterns and best practices.

Key Verbs: distinguish, analyse, differentiate, appraise, calculate, experiment, test, compare, contrast, create, design, setup, organise, manage, prepare.

Example: The following function will always return a correct result. True or False? Please justify your answer.

```
public int getMin(int[] data) {
    int min = data[0];
    for(int i = 1; i < data.length; i++) {
        if(data[i] < min) {
            min = data[i];
        }
    }
    return min;
}
```

Explanation: This is an analysis level question as the student is required to break down the code and understand the relationships and assumptions between each part. In this case, the student must realize that a zero length array or a null array will both cause an exception to be thrown.

3.4.5 Synthesis

Synthesis is defined as *putting together of elements and parts so as to form a whole*, in such a way as to create a program or a *program design not clearly there before*. This category recognizes *creative behavior* and student responses are expected to have a degree of variation and uniqueness.

Comprehension, application and analysis also involve the putting together of elements and the construction of meanings, but these tend to be more partial and less complete than synthesis in the magnitude of the task. Also there is less emphasis upon uniqueness and originality in these other classes than in synthesis.

Key Verbs: compose, plan, propose, design, formulate, arrange, assemble, collect, construct, choose, assess, estimate, measure.

Example: Write a program that will read in an arithmetic expression from the console and print out the result. For example, given the input $3*8/4+(6-(4/2+1))$, your program should output the answer 9 on a new line. The program should gracefully handle all exceptions.

Explanation: This is a synthesis level question as students could come up with many different correct implementations (e.g., using different tokenizer methods, recursive descent trees and other design patterns). The answers are expected to include a level of creativity.

3.4.6 Evaluation

Evaluation is defined as the *making of judgments* about the value of a program or program design. It involves the *use of criteria as well as standards for appraising* the extent to which the program or program designs are accurate, effective, economical, or satisfying. The judgments may be either quantitative or qualitative, and the criteria may be either those determined by the student or those which are given to him. Only those evaluations which are or can be made with specific criteria in mind are considered. Such evaluations are *highly conscious*; *require adequate comprehension and analysis* of the program or program design; and are primarily based on considerations of efficiency, economy, utility or specific means for particular ends.

Key Verbs: judge, appraise, evaluate, rate, compare, value, revise, score, select

Example: The function below is required to return the most frequently occurring character from a given input stream. You are a senior developer asked to review the implementation of this function as coded by a junior staff member. What comments would you make in regards to performance, correctness, assumptions, style and quality of the overall solution?

```
public char getMostFrequentChar(InputStream in) {
    //code implementation omitted
}
```

3.5 Interactive Examples

After reading the six category descriptions above, and self-rating their understanding of each, participants were then asked to classify some examples of examination questions. Participants had to provide answers, explanations and ratings on each of the twelve examples, such as the example seen in Figure 4. Participants were encouraged to scroll back to refer to the category definitions if needed.

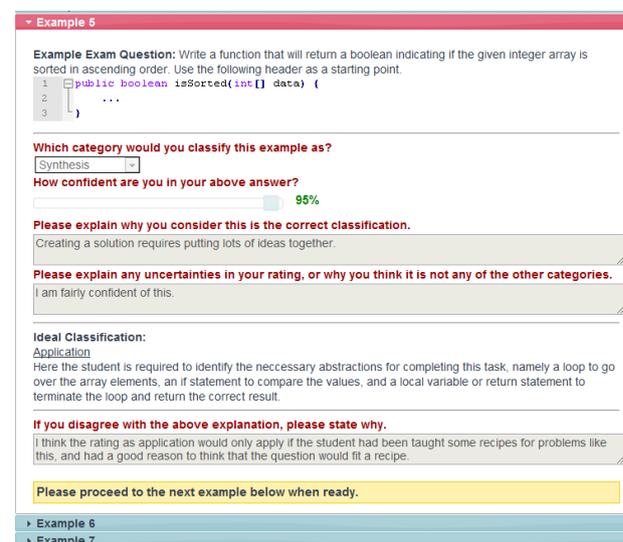


Figure 4: Tutorial Example Question

For each example, the participants classified the exam question on the Bloom scale. Participants were then required to self-rate their confidence in their classification, as well as to justify answers and comment on any uncertainties in their confidence. This was done in accordance with work by Chi M.T.H and colleagues (1994) showing that “Eliciting self-explanations improves understanding”.

The twelve example questions, and earlier category descriptions, were created by our Bloom expert - a computer science academic, one of this paper

’s authors, with an active research interest in the application of Bloom, SOLO and Neo-Piagetian frameworks to programming. Out of the twelve example questions, three were targeted as Knowledge, two as Comprehension, two as Application, two as Analysis, one as Synthesis and two as Evaluation. The unequal numbers were used so that participants would not be able to guess the last few by discerning the pattern and counting

answers. The order in which the questions were presented was randomized, but always in the same sequence for all participants. The following is a listing of these twelve examples and classification explanations (note that participants had to attempt classifying each example first before being shown the expected classification and explanation).

3.5.1 Example 1 - Application

Question: Write a function that will return a boolean indicating if the given integer array is sorted in ascending order. Use the following header as a starting point.

```
public boolean isSorted(int[] data) {
    ...
}
```

Explanation: Here the student is required to identify the necessary abstractions for completing this task, namely a loop to go over the array elements, an if statement to compare the values, and a local variable or return statement to terminate the loop and return the correct result.

3.5.2 Example 2 – Knowledge

Question: Circle the primitive data types in the following code snippet.

```
public boolean isSorted(int[] data) {
    boolean sorted = true;
    for(int i = 0; i < data.length; i++) {
        if(data[i] > data[i+1]) {
            sorted = false;
        }
    }
    return sorted;
}
```

Explanation: Here the student is required to simply recall the primitive data types as previously studied. This can be answered fully via memorization alone, without any need for understanding what a primitive data type is, or how it differs from other data types, or the differences between each of the eight data types. That is, if the student knows an `int` is a primitive data type, he can circle it in the code.

3.5.3 Example 3 – Evaluation

Question: A video rental store has implemented an online system where customers can login, browse through movies, select available movies, and rent movies online using a credit card. The video store has recently become concerned about security due to high profile events in the media. They have hired you as a security consultant to analyze their authentication and payment code and identify any vulnerabilities. Go through the code below and comment on its security. If not secure, why, and how should it be re-written to make it more secure?

Explanation: Here the student is required to comprehend and analyze the code by breaking it down into individual parts, then evaluate each part against security best practices. The student may identify buffer/integer overflows, SQL injection attacks, storing of plain-text passwords or weak encryption mechanisms, etc. The student can make a number of recommendations

on how to fix these. Note that even though the question uses the word `analyze`, this is actually an evaluation level task.

3.5.4 Example 4 – Analysis

Question: The following function takes an array of strings as inputs, and prints out each string and the number of times it appears in the array, in descending order. The code however throws a runtime exception when executed. Explain why. How would you fix it?

```
public static void printCount(String[] items) {
    Map counts = new HashMap();
    for(int i = 0; i < items.length(); i++) {
        int count = (int)counts.get(items[i]);
        counts.out(items[i], count+1);
    }

    String[] keys = (String[])
        counts.keySet().toArray(new String[0]);
    Arrays.sort(keys);
    for(int i = 0; i < keys.length; i++) {
        System.out.println(keys[i] + ": " +
            counts.get(keys[i]));
    }
}
```

Explanation: Here the student must first comprehend the question and intended behavior, then break down the function into logical parts (counting, sorting and printing), then identify if each part would operate as expected. The student should identify the bug in the counting loop in that the Map key values are not initialized and the code would throw a NullPointerException in line four. This could be fixed by prepending an if-not-contains-insert-key construct before line four.

3.5.5 Example 5 – Comprehension

Question: What is the output of the following code?

```
int[] data = {-3,6,9,2,4,15,-7,0};
int result = data[0];
for(int i = 1; i < data.length; i++) {
    if(data[i] > result) {
        result = data[i];
    }
}
System.out.println(result);
```

Explanation: Here the student is required to read the code, interpret the individual parts, understand what it does, and trace the execution to derive the right answer. To do this the student requires knowledge of programming syntax, control structures and variable scope as a pre-requisite.

3.5.6 Example 6 – Comprehension

Question: What is the output of the following code snippet?

```

int a = 3;
int b = 7;
int c = 0;
int[] data = {1,6,5,2,3,9,6,3,1,6,8,0,3};
for(int i = 0; i < data.length; i++) {
    if(data[i] > a || data[i] < b) {
        c++;
    }
}
System.out.println(c);

```

Explanation: Here the student is required to read the code, interpret the individual parts, understand what it does, and trace the execution to derive the right answer. To do this the student requires knowledge of programming syntax, control structures and variable scope as a pre-requisite.

3.5.7 Example 7 – Knowledge

Question: The javac.exe command is used to compile java code. True or False?

Explanation: Here the student is required to simply recall the function of the javac.exe command, or which command is used to compile java code. This can be done by rote memorization without any further understanding of the compilation process or other java internals.

3.5.8 Example 8 – Synthesis

Question: A video store has a list of movies in a CSV file with the following header: "movie title", "year released", "genre", "main actor/s", "rating (1-5)". The main actor/s field can contain a single name, or multiple names separated by a comma. An example line is:

"The Social Network", "2010", "Jesse Eisenberg, Andrew Garfield, Justin Timberlake", "4.5"

The store has hired you to write a command-line program that will return the top three most popular actors in each genre (i.e., highest average ratings of all movies they appeared in in that genre). Assume the path to the CSV file is passed in as the first command-line argument.

Explanation: Here the student must comprehend and analyze the scenario, then apply the correct programming abstractions to parse the CSV and process the data to derive the correct answer. There are multiple ways of implementing this correctly, and the task description leaves students open to use some creativity in coming up with unique solutions.

3.5.9 Example 9 – Knowledge

Question: Write a SortedHashMap implementation. As discussed in lectures and practised in tutorials, the sorted map should expose two iterators: one that loops through all key/value pairs sorted in ascending key order, the other that loops through all key/value pairs sorted in ascending value order. The Map should work with any object that implements Comparable.

Explanation: This would be a non-trivial problem which could be solved in many different ways if the student had never come across a SortedHashMap before. However, the question states ``as discussed in lectures and tutorials``, which implies the student has had sufficient practice at this exercise. So even though this

could be a complex problem with unique solutions (i.e., Synthesis), since the students have had significant prior practice at this exact problem, it is actually a Knowledge question as it can be completed via rote memorization.

3.5.10 Example 10 – Analysis

Question: An employee at your company writes the following function, which takes a java InputStream as a parameter and returns the average word-count of sentences. The function sometimes returns incorrect results however. Why? How would you fix it?

```

public double avgWordCount(InputStream in) {
    ...code omitted from tutorial...
}

```

Explanation: Here the student must first comprehend the question and intended behavior, then break down the function into logical parts, and identify if each part would operate as expected. This is not evaluation however, as the student is not asked to comment on the algorithm implemented, but rather to break down the algorithm to find the incorrect assumption that leads to the bug.

3.5.11 Example 11 – Application

Question: Fill in the missing code in the following function that calculates and returns n! (factorial).

```

public long factorial(int n) {
    long result = 1;
    ...write missing code here...
    return result;
}

```

Explanation: Here the student is required to identify that the use of a loop is needed to compute the right answer, without being hinted of this. This is assuming that the student has not rote memorized the code for n!, in which case this would be a knowledge question.

3.5.12 Example 12 – Evaluation

Question: You are a senior developer in a company that creates iPhone games. A junior developer is tasked with creating a love score calculator which takes two names as input and returns a compatibility rating score based on the following rules [omitted]. The junior developer submits the following code as a solution. You are tasked with reviewing the code for quality, correctness, efficiency and style. What comments would you make and why? Explain in as much detail as you can.

```

public static int calcComp(String n1, String n2) {
    ...code omitted from tutorial...
}

```

Explanation: Here the student is required to comprehend and analyze the code by breaking it down into individual parts, then evaluate each part against a series of metrics. The student may identify a number of potential bugs (divide by zero, integer overflow), identify unspecified assumptions (treatment of space and other special characters, treatment of repeating characters), suggest graceful error handling, suggest better named variable names, etc.

3.6 Post-Survey

Upon completing all twelve example classifications, participants were asked to complete a two-question post survey:

1. Based on your new understanding of the Bloom Taxonomy, how confident are you in being able to correctly classify programming exam questions at the correct level (out of the six possible levels)?
2. Did you find this tutorial useful and effective in increasing your understanding of the Bloom Taxonomy? Why or not?

4 Results

A total of ten participants completed our interactive tutorial and example classifications. These consisted mostly of computer science tutors and one computer science professor. The average participant Initial Confidence (measured in the pre-survey) score was 30.8% (sd. 23.36) (only 2 participants self-rated at 70%, and one at 40%, while the rest self-rated at 30% or lower). Prediction Confidence (measured during the initial reading of the category descriptions) ranged between 67 and 71%. The results after participants had classified all of the twelve examples are summarized in Figure 5.

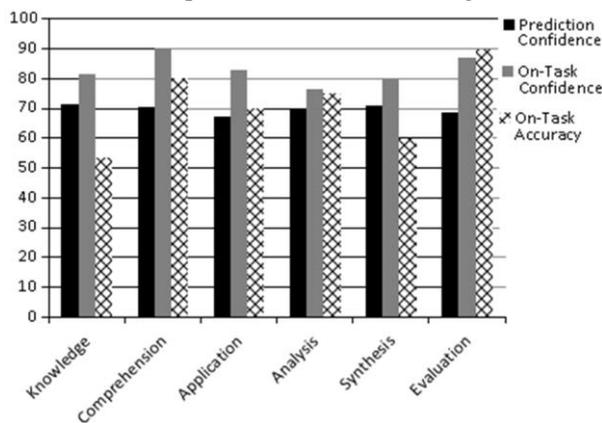


Figure 5: Prediction Confidence, On-Task Confidence and Accuracy

This chart shows the six Bloom categories along the horizontal axis. Each category is sub-divided into three columns. These are, from left to right, Prediction Confidence, On-Task Confidence and On-Task Accuracy (i.e., confidence for each category after reading each description in the introduction, confidence given in each interactive example, and number of times when participant classifications matched the expected expert classifications). The On-Task Accuracy is the percentage of participants who agreed with our expert classification for each question. For the purposes of evaluating the tutorial system, we treated the expert's classifications as the correct (or expected) classification.

4.1 Participant Responses to the 12 Examples

The following is a listing of the participant classifications and comments for each of the twelve examples.

Example 1: Nine participants classified this as Application (the expert classification). One participant however argued that this was a Synthesis task as “creating

a solution requires putting lots of ideas together”. This participant expressed a 95% confidence in this answer, and disagreed with our explanation as to why our expert classified it as Application. The participant wrote “I think the rating as application would only apply if the student had been taught some recipes for problems like this, and had a good reason to think that the question would fit a recipe.”

Example 2: Nine participants classified this as Knowledge (the expert classification). One participant classified it as Comprehension as “the student is required to understand and trace through the code to circle the primitive data types”, but subsequently agreed with our expert's explanation.

Example 3: Eight participants classified this as Evaluation (the expert classification). Two participants labelled it as Analysis as “It requires the student to look through the code snippet and comment on what security flaws it has”. These two participants did not specify if they subsequently agreed with our expert classification.

Example 4: Eight participants classified this as Analysis (the expert classification), but two picked Comprehension. One of these two wrote “It could be in analysis, but it asks how it would be fixed which wouldn't suggest it's an analysis”. It seems this participant took a strict definition of the term analysis and assumed that since the student had to fix/write code, then it had to be a different category. Both subsequently agreed with our expert classification.

Example 5: Nine participants classified this as Comprehension (the expert classification). The other marked it as Knowledge, stating “This code seems to be absolutely standard, so the student should have seen it often, and know what it does. (find the max) They then just have to look at the input (I assume that knowing the max of numbers is trivial at this level)”. This participant assigned a 50% confidence score to this answer, and justified this as “If the student actually traces the code, that would be application in my view.” After being shown our expert's classification and explanation, the participant commented “As noted, I think this code would be very familiar, and not need to be broken down and treated as parts.”

Example 6: Seven participants classified this as Comprehension (the expert classification). Two marked it as Application and one as Analysis. One participant justified the choice of Application with “As noted, I see tracing as requiring more than knowledge; it requires complex abstractions of the machine model, and using them on the code that is given.” Another participant commented that “it could be comprehension” and the other participant believed that tracing had to be Application.

Example 7: Seven participants classified this as Knowledge (the expert classification). One participant classified it as Analysis and one as Application. The comments left by these two did not adequately describe their reasoning. The last participant labelled it as Comprehension, stating “the student is required to understand the how to compile java program”. This suggests a misreading of the question or incorrect

assumption about the knowledge required to answer the question.

Example 8: Six participants classified this as Synthesis (the expert classification). One picked Analysis and stated that *"Could be anything from Application up to Synthesis but I'm guessing somewhere in between"*. The other three participants all marked it as Application. One stated that *"I assume that the student knows well the available commands, and just has to adjust the flags and then string them together in the right order ... if actual problem solving was needed, i would rate it as synthesis."* Another participant stated *"implementation questions most likely are application ... not an analysis, nor evaluation as it doesn't ask for the students opinion"*.

Example 9: No participants classified this as Knowledge (the expert classification). Instead, four picked Synthesis, five Application and one Analysis. From inspecting the comments, the two main reasons for this were either that the participants *"missed the clue that the student had seen this exact problem"* or assumed that *"this question is hard for students who are in knowledge level"*. This second comment implies that a student in the Knowledge level cannot rote learn complex solutions. Five participants agreed with the expert classification after being shown the justification. The rest either disagreed on the basis mentioned above or did not elaborate further.

Example 10: Seven participants classified this as Analysis (the expert classification). Three however marked it as Evaluation. One stated *"I am evaluating another colleague's code, which requires judgements in order to apply fixes to it"*. Another commented *"It could be just analysis, but it's also working out what might be wrong with the code and suggesting alternatives."* The third incorrect participant stated *"too much abstraction"*.

Example 11: Five participants classified this as Application (the expert classification). One marked it as Knowledge and stated *"If the student hasn't memorized this, I would see it as synthesis, unless they have a pattern that they know would be used to solve the problem. Only if they know which pattern to use is application the suitable level."* One participant labelled it as Analysis with 100% certainty, stating that *"It requires that I break the problem down"*. Two participants labelled it as Synthesis because *"students could come up with many different correct implementations"*. One participant marked it as Comprehension because *"the student needs to comprehend the code before answering"*.

Example 12: All twelve participants classified this as Evaluation (the expert classification).

4.2 Final Confidence and Participant Feedback

After completing the twelve examples, the average participant Final Confidence score was 75% (sd. 11.55), an increase from the 30.8% Initial Confidence before starting the tutorial. All participants responded positively in the final feedback question. The common trend in these comments was that the category descriptions were good for gaining a basic grasp, but the interactive examples with justified answers were very useful in consolidating their understanding.

5 Discussion

Most of our participants had very little exposure to Bloom prior to taking the tutorial, hence the low average Initial Confidence of 30.8%. The intention of the tutorial was to be quick but sufficient for participants to be able to apply Bloom to Programming Fundamentals; hence the small set of simple examples for each Bloom category. After completing our tutorial however, the Final Confidence average increased to 75%. Participants however still did not feel entirely confident in being able to do this consistently. An analysis of the self-reflection ratings is presented and discussed in greater detail in Gluga and colleagues (2012).

Overall the tutorial was successful in training participants on how to apply Bloom to programming questions. Participants that had little confidence prior to the tutorial came out with a much higher understanding of Bloom. However, the evaluation confirms ambiguity in the interpretation of the Bloom categories due to dependence on knowledge of the learning context and due to different assumptions made by participants.

Example 9 demonstrates a challenging case. The information supplied indicates that the learners are very familiar with the algorithm and associated code. However, it is unlikely students would rote-memorize the amount of code needed to answer this question. While our expert coded this as Knowledge level, it may be more likely that a student may need to recall the pseudo-code or algorithm and translate it into the appropriate syntax to derive the answer. This would place the task at the Comprehension level (translating from one form into another, using recalled abstractions). The role of such questions in future versions of the system will be reviewed. Perhaps the role of such a question could be to highlight how the difficulty in coding the Bloom level may point to problems with the question.

Evidence showed that some participants also confused the literal definition of the Bloom category labels with the classification of some questions. More example exercises for each category type may have been useful to indicate whether these participants learnt to apply the categories with greater consistency.

6 Conclusion and Future Work

There is growing recognition that computer science educators are over-estimating the ability of CS1 students and are often setting programming-fundamentals assessment exercises that are overly-ambitious in the number of concepts and level of competence tested (Lister 2000 and 2001, Oliver 2004, Petersen et. al. 2011). The aim of this paper was to explore ways to measure learning progression in the programming fundamentals sequence of subjects, to ensure that content is taught and assessed at an appropriate level of competence.

The Bloom Taxonomy is a framework for specifying the sophistication of learning objectives, which is already part of the ACM/IEEE CS Curriculum (2008) specification, and can be used as a tool to classify the competence level of assessment questions. Bloom is often not well understood however. We thus created an interactive tutorial to train computer science educators on

how to apply Bloom in classifying programming questions and evaluated the results.

The evaluation showed that the tutorial was effective in developing participants' confidence in identifying the level of performance involved in programming exam questions. The evaluation also confirmed previously documented ambiguities in the application of Bloom to cases where knowledge about the learning context is required for accurate classification. Participant feedback comments at the same time revealed other reasons as to why consensus is not always reached, namely due to pre-conceived misunderstandings of the categories, or different interpretations about the complexity of tasks and sophistication required to solve them.

The study had 10 participants and all gained confidence in using Bloom to classify assessment tasks. These results are promising and the qualitative results will be of value for informing refinements to the interface and examples. It will then be important to evaluate with a larger number of participants, including people who actually design curricula and design exam questions. Additionally, the experiment can be repeated with a different cognitive development framework, e.g., Neo-Piagetian Theory, to see how this compares to Bloom in terms of classification consistency and user satisfaction.

We believe that our ProGoSs system's Bloom tutorial is the first such system that helps teachers of programming fundamentals have greater understanding of Bloom, as a foundation for more systematic design of teaching and learning materials and assessment of how well student learning meets the intended goals. An enhanced version of this tutorial may soon be made openly available online.

Acknowledgements

We would like to thank the Smart Services CRC who is partially sponsoring this project and all our colleagues that we have collaborated with throughout the project.

7 References

- Anderson, W.L., Krathwohl, D. R. (Eds.). Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., Raths, J. and Wittrock, M. C. (2001). A taxonomy for learning, teaching and assessing: A revision of Bloom's Taxonomy of Educational Objectives. New York: Allyn Bacon Longman.
- Anderson, W.L., Sosniak, A.L, Bloom, B.S. (1994). Bloom's taxonomy: a forty-year retrospective. Chicago: University of Chicago Press
- Biggs JB & Collis KF (1982) Evaluating the quality of learning: the SOLO taxonomy (Structure of the Observed Learning Outcome). New York, Academic Press.
- Bloom, B.S. (Ed.) (1956) Taxonomy of Educational Objectives: Handbook I: Cognitive Domain, Longmans, Green and Company.
- Centre for the Advancement of Teaching and Learning, University of Western Australia (2005). A Basic Guide to Writing of Student Outcome Statement. Retrieved 24 August 2011 from http://www.catl.uwa.edu.au/current_initiatives/obe/outcomes
- Centre for Learning and Professional Development, University of Adelaide (2011). Writing learning objectives. Retrieved 24 August 2011 from <http://www.adelaide.edu.au/clpd/curriculum/objectives/>
- Chi, M.T.H., Leeuw, N.D., Chiu, M.H., Lavancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive Science*, pages 439-477. Vol. 18.
- Computer Science Curriculum 2008: An Interim Revision of CS 2001. Association for Computing Machinery and IEEE Computer Society, 2008. <http://www.acm.org/education/curricula/ComputerScience2008.pdf>
- Computer Science Curriculum 2013, Association for Computing Machinery and IEEE Computer Society, <http://www.sigart.org/CS2013-EAAI2011panel-RequestForFeedback.pdf>
- Gluga, R., Kay, J., and Lever, T (2010). Modeling long term learning of generic skills. In V. Aleven, J. Kay, and J. Mostow, editors, ITS2010, Proceedings of the Tenth International Conference on Intelligent Tutoring Systems, pages 85-94. Springer, 2010.
- Gluga, R., and Kay, J. (2009) Largescale, long-term learner models supporting flexible curriculum definition. In Proceedings of the Workshop on Scalability Issues in AIED, held in conjunction with AIED2009, pages 10-19, 2009.
- Learning and Teaching Centre, Macquarie University (2008). Writing learning outcomes. Retrieved 24 August 2011 from http://www.mq.edu.au/ltc/about_lt/assess_docs/writing_learn_out.pdf
- Lister, R., (2000), On Blooming First Year Programming, and its Blooming Assessment. In Proceedings of the Fourth Australasian Computing Education Conference (ACE2000), Melbourne, pages 158-162.
- Lister, R., (2001) Objectives and Objective Assessment in CS1. Proc. SIGCSE Technical Symposium on Computer Science Education, Charlotte NC, USA, pages 292-296, ACM Press.
- Lister, R. (2011). Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. Australasian Computing Education Conference (ACE2011), pages 9-14. Vol. 114.
- Morra, S., Gobbo, C., Marini, Z. and Sheese, R. (2007) Cognitive Development: Neo-Piagetian Perspectives. Psychology Press.
- Oliver, D., Dobeles, T., Greber, M., and Roberts, T. 2004. This course has a Bloom Rating of 3.9. In Proceedings of the Sixth Australasian Computing Education Conference (ACE2004), pages 227-231, Vol. 30.
- Petersen, A., Craig, M., and Zingaro, D. (2011). Reviewing CS1 Exam Question Content. In Proceedings of the 42nd ACM technical symposium on Computer science education (SIGCSE2011), pages.631-636
- Sheard, J., Carbone, A., Lister, R. and Simon, B. (2008), Going SOLO to assess novice programmers. In Proceedings of the 13th annual conference on Innovation and technology in computer science education (ITiCSE2008), pages 209-213, Vol. 40.
- Thompson, E., Luxton-Reilly, A., Whalley, J.L., Hu, M., Robbins, P. (2008). Bloom's Taxonomy for CS Assessment. In Proceedings of the Tenth Conference on Australasian Computing Education (ACE2008), Vol. 78
- Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P.K.A. and Prasad, C. (2006). An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. In Proceedings of the Eighth Australasian Computing Education Conference (ACE2006), pages 243-252, Vol. 165.
- R.Gluga, J. Kay, R.Lister, S.Kleitman, T.Lever. Over Confidence and Confusion in using Bloom for programming fundamentals assessment. Special Interest Group in Computer Science Education, 2012 (SIGCSE2012)

An exploration of factors influencing tertiary IT educators' pedagogies

Sally Firmin¹, Judy Sheard², Angela Carbone³, and John Hurst²

¹School of Science, Information
Technology and Engineering
University of Ballarat
PO Box 663, Ballarat 3353,
Victoria
s.firmin@ballarat.edu.au

²Faculty of Information
Technology
Monash University
Victoria, 3800
Judy.Sheard@monash.edu,
John.Hurst@monash.edu

³Office of the Pro Vice-
Chancellor (Learning and
Teaching)
Monash University
Victoria, 3800
Angela.Carbone@monash.edu

Abstract

This paper presents factors that influence and shape tertiary IT educators underpinning teaching philosophy. This work is the first part of a larger project investigating ways tertiary IT educators think about their teaching and develop their practice focusing on experiences and influences of technology, and the emergence of digitally based pedagogies. A qualitative grounded theory approach utilised semi structure interviews as the data source. Preliminary investigation identified four emergent themes from the data. The theme, 'pedagogical foundations' is explored in this paper. This theme provides details of tertiary IT educators' underpinning ideals, values and philosophy of teaching, grounded by thoughts, reflections and comments of their experiences. Exploring these ideas can improve the quality of teaching, better utilise new and emerging technologies, and nurture contemporary student-centred learning environments.

Keywords: pedagogy, IT academics, teaching philosophy, qualitative research, grounded theory.

1 Introduction

Pedagogy is a complex and vague term. Its meaning has been the source of debate in the discourse of many educators over recent times. This is partly due to the volatile higher education teaching and learning environment within which tertiary educators currently operate. This changing environment has manifested a shift from the traditional didactic teacher-focussed approach, to a technology-enhanced student-centred collaborative approach.

Developing an understanding of pedagogy and the factors influencing its formation can help to improve teaching practice. Harris (2005) claims that most academics (apart from those in education schools) do not have a background or formal training in education theory and pedagogy. Many educators are overwhelmed by the complexity of pedagogy (Ramsden, 2003). Harris (2005) found that by introducing academics to education theory and pedagogy, such as Bloom's taxonomy, the academics were better equipped to begin improving teaching and

learning outcomes. These authors promote the notion that an awareness of teaching philosophy can better equip educators in a tertiary educational context.

There are many factors influencing pedagogy formation. This provides the impetus for the research reported in this paper. By unravelling the notion of pedagogy and its development, through research, it is proposed that we can build an understanding of factors that may help improve teaching practice. The work of Ramsden (2003) and Harris (2005) provide supportive evidence for the importance of this research.

An additional and important consideration is that various disciplines have different pedagogies. Disciplines such as medicine and law have reported distinctive pedagogies. Shulman (2005) terms these *signature* pedagogies. In our research, we investigate current pedagogical influences of tertiary IT educators.

This paper reports on phase one of a two-phase study. Phase one of the study focuses on factors that influence tertiary information technology (IT) educators' pedagogical development. Phase two of the study will investigate tertiary IT educators' experiences of using technologies in teaching and the role technology plays in shaping their pedagogies. The specific research question being investigated in phase one is:

How do tertiary IT educators develop their pedagogy?

The structure of this paper is as follows. Section 2 reviews the literature relating to defining pedagogy, and factors influencing its development. Section 3 gives a justification for the use of the grounded theory research approach. This is followed by a detailed description of the implementation of a Straussian Grounded Theory (GT) study in Section 4. The results of this study are presented in Section 5, discussed in Section 6 and followed by conclusions in Section 7.

Four categories emerged from the GT analysis. The category which describes teachers underpinning ideals, values and philosophy of teaching, and their thinking behind the practice, is discussed in this paper.

2 Pedagogy Research

The aim of this section is to examine current research pertaining to pedagogy in a tertiary education environment. The literature shows that over time educators' views of the concept of pedagogy have become more complex and show a divergence from teacher-directed instruction to student-centred learning. The context for this research is a tertiary education

setting; however, other educational levels will be encompassed in an informative basis. Understanding the evolution of pedagogy will assist in providing us with pedagogical themes and identify the development journey.

2.1 A Review of Pedagogy

Pedagogy is a complex, misunderstood, ill-defined word, with its meaning evolving overtime (Canning, 2007). According to Beetham and Sharpe (2007) “despite its etymological connection with children (*paidia*), contemporary use of the term has lost its exclusive reference to childhood while retaining the original sense of leading or guiding to learn” (p. 1). Academics have needed to alter their thinking and recognise how pedagogical concepts and practices have altered (Schilb, 1999). For centuries the pedagogy of the classical curriculum was a dry and sterile pedagogy of grammar instruction, whereas contemporary thinking is one of ideas, values, critical thinking, moral deliberation, and logical reasoning (Gregory, 2001).

Historically pedagogy has been associated with the teaching of children as its background emanates from the Greek word ‘paid’, meaning child, and ‘agogus’ meaning leader of (Conner et al., 1996). As defined by Smith and Lowrie (2002) pedagogy refers to the teacher’s relationships with children. More explicitly, it refers to “appropriate ways of teaching and giving assistance to children and young people” (Loughran, 1999, p. 14). Traditional notions of pedagogy were associated with teacher-centred instruction (Conner, et al., 1996). This is thought to have originated from the Calvinists who believed wisdom was evil. They advocated adults monitor, control, and restrict children’s learning to keep them innocent (Conner, et al., 1996). In this traditional pedagogic model, teachers held responsibility for making decisions about what will be learned, how it will be learned, and when it will be learned. Teachers directed the learning (Conner, et al., 1996).

Contemporary definitions describe pedagogy as the art, profession or science of teaching (Beetham & Sharpe, 2007; Chapuis, 2003). Pedagogy is often represented as the philosophy and instructional approaches associated with good teaching (Kemmis & Smith, 2006). However, pedagogy is often seen as a nebulous concept, with some educators using it as a synonym for teaching (Conner, et al., 1996), but pedagogy means more than teaching. As reported by Ladwig and King (2003) pedagogy is about how teaching is done rather than what is taught. Pedagogy is about the teaching and learning activities teachers use and how they assess their students’ progress. Smith and Lowrie (2002) also support this concept and indicate that pedagogy can be an effective way of describing the relationships between teaching, learning and assessment in classrooms, they believe to talk of pedagogy is to talk of the appropriate ways teachers interact with learners. Beetham and Sharpe (2007) argue that some educators are still at odds with the emphasis on teaching, with their preference on the activity of learning, suggesting that in a learner-centred environment teaching should not be the focus of concern.

Contemporary writers suggest that the traditional teacher-centred view of pedagogy is not only becoming

student-centred but more complex. Mortimore (1999) contends that academics’ and researchers’ notions of pedagogy have become more complicated over time. He argues that a deepening in our understanding of cognition and meta-cognition have influenced the conceptualisation of pedagogy. He describes the current model of pedagogy as being a complex one which includes relationships between the teacher, learning context, content, and learning. Chapuis (2003) suggests pedagogy requires a broad repertoire of strategies and sustained attention to what produces student learning in a specific context. Smith and Lowrie (2002) believe pedagogy embodies “the relational, emotional, moral and personal dimensions of the teaching and learning process” (p. 6). Whilst Waters (2005) endorses pedagogy as encompassing both formal and informal knowledge about teaching and learning and is reliant on both the learner and the teacher.

These authors all provide evidence of a growing conception of what pedagogy embodies. Note the gradual change from teacher-focused to student-centred learning, and the co-relationship between educator and student. It is this expanded, broader vision encompassing learning, relationships and student-centredness that will underpin this research.

2.2 Factors Influencing Pedagogy

Some of the factors that influence educators’ pedagogies include perception of teaching and learning roles, folk pedagogies, personal learning experiences, educational technology, government and institutional policy, peer evaluation, reflective practices, student evaluations, teaching and learning context, and, understanding of teaching. A brief overview of each follows.

The educators’ perspective of the teaching role is an important factor in determining teaching approach. Biggs (2007) suggests these are divided by who is in major control – the teacher or the student. These roles have been characterised in educational language as ‘Sage on the stage’ (teacher-directed) and ‘Guide on the Side’ (student-directed) (King, 1993). Biggs suggests that each approach results in very different engagement from the learner. Furthermore, the way educators have been taught is likely to have an impact on the way they teach. According to Shulman (2004) educators own learning experiences influence their approach to teaching.

Educators’ existing beliefs about teaching influence their approach (Raths, 2001). These beliefs have been termed ‘folk pedagogies’. According to Olson and Bruner (1998) folk pedagogies are lay theories or intuitive beliefs teachers have about the way students learn.

Educational technology has played a significant role in shaping many educators contemporary pedagogies. Newson (1999) coined the term techno-pedagogy describing it as models of teaching and learning associated with instructional technology. The notion of technology-enhanced teaching shows a shift in teacher’s role from controller to coach of learning (Jonassen, Howland, Marra, & Crismond, 2008).

Government and institutional policies have been reported as influencing factors on tertiary IT educators’ teaching approaches. Tutty, Sheard and Avram (2008) reported a lack of support and encouragement for IT academics, restricting them with teacher-centred policies

which are counter to their preferred student-centred styles.

Peer evaluation and observation can provide educators with useful commentary about the quality of course content, structure, and assessment (Bain, 2004). Carbone and Kaasbooll (1998) found that peer observers could also offer feedback on teaching based organisational and communication issues providing a chance for educators to reflect and compare without the pressure of performance. Ladwig (2005) suggests that peer review can provide analysis and thinking at a pedagogical level and that this process can lead to improved educational outcomes.

Reflective practice influences the different ways educators think about teaching and function as teachers. Burn et al (cited in Marsh, 2008) found that critical self reflection is an essential tool for teachers to utilise as it helps them undertake informed action and provides a rationale for practice. Ramsden (2003) found that just thinking about teaching is not enough, the challenge is to merge the thinking and doing. Ramsden found this could have likely implications for student learning outcomes.

Evaluation tools can provide educators with an opportunity to reflect on the quality of their teaching. Kaplan (cited in Bain, 2004) suggests that by asking students the right questions, their answers can aid educators to make judgements about the quality of their teaching. Bain stresses that the student ratings are not by themselves evaluations.

Educational learning spaces are complex busy environments in which varying groups of students must be organised. Teachers require a highly developed ability to manage these complex situations, multiple activities and unpredictable events (Doyle cited in Mortimore, 1999).

Finally, educators' knowledge of teaching can influence pedagogical development. University teaching is very complex and Ramsden (2003) proposes that most educators feel they have a better grasp on its complexities than they actually do. There are increasing demands on educators in terms of teaching skills (Biggs, 2007). Traditional approaches no longer work with a much more diverse student population. Biggs believes a fresh look at teaching is necessary.

2.3 Tertiary IT Education Context Gap

There is wealth of literature in the education field about tertiary teaching pedagogy, but a scarcity of work in the discipline of IT, particularly in an Australian context. There are a few studies that provide examples of teaching experiences of IT educators (Lister et al., 2007) but few were found that have investigated factors influencing tertiary IT educators' pedagogical development.

The work reported in this paper is inspired from published work of several studies within the IT discipline. These include work on scholarship pursuits of IT academics by Lynch, Sheard, Carbone and Collins (2005) and, work by Tutty, Sheard and Avram (2008) which presented a model of IT academics teaching experiences. Of significant influence is work by Kutay and Lister (2006), whose research aimed at facilitating a community of practice to foster ways of discussing pedagogy in a higher education IT school, and the work of Lister et al

(2007) who investigated ways tertiary IT educators understand teaching.

This section provided an outline of the background literature regarding theory and factors influencing pedagogy. The next section provides a brief outline of the research design approach.

3 Research Design

The aim of this section is to provide a description and justification of the research design approach adopted in this study.

3.1 Qualitative Inquiry

A qualitative approach has been adopted for this work. Qualitative inquiry is typically used for the exploration of social phenomena or situations in which individuals are involved with various types of processes, such as educational processes (Hazzan, Dubinsky, Eidelman, Sakhnini, & Teif, 2006; Myers, 1997). In the context of this research factors, influencing the development and formation of tertiary IT educators' pedagogies is the phenomenon being investigated.

3.2 Interpretivism

An interpretive philosophical view underpins the work done on this project. Interpretivism is a view that cultures can be understood by examining peoples' beliefs, their ideas, and the meanings that are significant to them. All knowledge is a matter of interpretation (Crotty, 1998). Interpretivism is an appropriate choice because tertiary IT educator's knowledge of their world is formed through their teaching and learning experiences (epistemology).

3.3 Grounded Theory Methodology

A GT approach will be modelled on this project. According to Strauss and Corbin (1990) "A grounded theory is one that is inductively derived from the study of the phenomenon it represents. That is, it is discovered, developed, and provisionally verified through systematic data collection and analysis of data pertaining to that phenomenon" (p. 23).

GT is appropriate for research studies when all the concepts pertaining to the given phenomenon have not been identified in a particular context (Strauss & Corbin, 1990), as is the case with this project.

A review of the genealogy of GT reveals the following major approaches:

- Glaserian GT: Glaser and Strauss (1967) and Glaser (1978)
- Straussian GT: Strauss (1987) and, Strass and Corbin (1990, 1998, 2008)

The Straussian GT approach was chosen for this project as it best matched the parameters of the research, as follows: it allows the researcher to enter the research field with preconceived ideas, a predetermined problem statement, extensive review of the literature, and an interview protocol (Charmaz & Bryant, 2007). The researcher had previously undertaken all of these tasks.

GT has been used in many discipline areas. Within the context of IT education, GT has been used in a number of studies. For example, it was used as a research

methodology to investigate IT student capstone projects (Kollanus & Isomottonen, 2008a). GT was used by Kollanus and Isomottonen (2008b) in several studies on using test driven development in extreme programming. There are also several notable studies by Kinnunen and Simon (2010) and Dunican (2006) who used GT to investigate the learning and teaching of novice computer programming.

While this section provided an explanation and justification of the research approach used in this study, the following section provides details of the implementation of the GT approach.

4 Implementation of GT

4.1 Research Approach

This section provides an explanation of the implementation of phase one of the two-phase study, applying the theoretical approach discussed in the previous section. A detailed description of the first two coding stages of GT is presented. This explains the research techniques used and in doing so illustrates the Straussian GT approach. Data gathered through phase one is presented in section 5 results.

The Straussian model of GT data collection and coding processes was applied in an integrative iterative fashion during phase one of this project. This approach was based on modelling presented by McNabb (2010, p. 256) and Hoda, Nobel and Marshall (2010, p. 1). See adaptation in Figure 1.

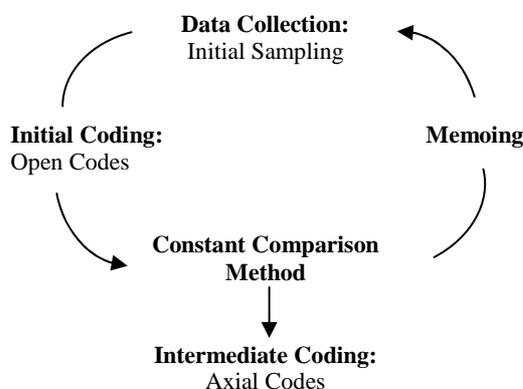


Figure 1: Phase 1 Grounded Theory Approach

4.1.1 Data Collection: Initial Sampling

Data collection in phase one consisted of four one-hour interviews with tertiary IT educators from a regional Australian university. The approach taken followed the work of Golding (2007) who advises initial data sampling in GT studies be conducted openly with a broad section of participants. Participants were recruited based on a range of demographic characteristics such as gender, age, teaching experience, and year level.

Four academics, a male and female with over 40 years teaching experience and a male and female with 10-20 years teaching experience were selected. All academics had experience in teaching both undergraduate and post graduate programs.

The interview protocol consisted of eight questions divided into two sections. The first section was designed to build a profile of the teacher, and gathered information concerning, mentors, teaching career highlights, perceived characteristics of good teachers, course preparation, assessment and delivery. Questions included:

- How did you get into teaching?
- Can you describe key moments, experiences or people that have influenced your teaching philosophy?
- Can you think of any other factors (e.g. circumstances) that have influenced your teaching?
- What are the most important characteristics you believe a good teacher must have?
- How do you go about teaching a course?

These questions gathered information aimed at answering the research question, which is the focus of this paper: How do tertiary IT educators develop their pedagogy?

Data from the second section of the interview protocol will be further developed in phase two of this project, and is outside the scope of this paper.

4.1.2 Initial Coding: Open Codes

After data collection, the initial coding process was undertaken, an *open coding* approach, consistent with Straussian GT was implemented. According to Strauss and Corbin (1990) open coding is “the process of breaking down, examining, comparing, conceptualizing, and categorizing data” (p. 61).

The open coding process in this study consisted of several iterations. A sentence-by-sentence technique was utilised. Each interview was coded with the previous interview in mind, this is known as a *constant comparative approach* (Glaser & Strauss, 1967). The first interview transcript was coded in a sequential fashion. Subsequent interview transcripts were coded in an iterative fashion using the constant comparative method to revisit, revise and identify additional codes.

4.1.3 Memoing

The process of memo writing was conducted in parallel with the data collection, coding and constant comparison method. In this project, consistent with the work of Charmaz (2006), memo writing was undertaken in order to capture thoughts containing analysis, comparisons, connections about codes, and categories or relationships which link the categories. Continued writing of memos throughout the research process assisted to elevate the level of abstraction of ideas, and codes began to stand out and take shape into theoretical categories. Memos developed in MS Word were chronologically dated and themed for efficient future comparison, reflection and retrieval.

4.1.4 Intermediate Coding: Axial Codes

The final step in phase one was the intermediate coding phase. During this phase, *axial coding* was completed. Axial coding, as defined by (Strauss & Corbin, 1990, p. 96), is “a set of procedures whereby data are put back

together in new ways after open coding, by making connections”. The axial coding process was conducted using the following steps, commencing with the grouping of open codes (identified during initial coding) into axial codes. Axial codes were then grouped into emergent *categories* (themes). Each axial code was further defined through the identification of *dimensions* and *properties* and the development of a *paradigm model*.

The axial coding process was used to extend the analytic work of initial coding and strategically reassemble fractured data into emergent *categories*. Each axial code was deconstructed into a number of *properties*. Strauss and Corbin define properties as “attributes or characteristics pertaining to a category, and *dimensions* are the location of properties along a continuum (Strauss & Corbin, 1990). Dimensions will be uncovered in phase two of the project, as more data is collected and the constant comparative method applied. For details of the axial codes and properties generated in this study, refer to section 5.2.

As part of the axial coding process a paradigm model was developed. Paradigm modelling is recommended by Strauss and Corbin (2008) to be useful in providing answers to questions of context and developing insight into a phenomenon (Dunican, 2006). In this project it was used to deconstruct and reframe data uncovered in the ‘pedagogical foundations’ category. It was used to gain insight into pedagogical development in terms of the conditions in which pedagogy is reflected upon, by the interactions causing the reflection, and, the resultant change. For details of the paradigm model developed in this study, refer to section 5.2.

During the proposed second phase of the project, the GT process will be continued, utilising a theoretical sampling data collection technique. The selective coding approach will be applied iteratively and integrated with the GT memoing technique and constant comparison method, until data saturation point is reached, at which point memos will be sorted and a theory developed.

This section has provided a description of the practical application of GT implemented in phase one. The next section will provide details of results of phase one, in particular, the category ‘pedagogical foundations’, which emerged from this analysis.

5 Results

This section provides results of the analysis conducted in phase one, in particular, a description of the emergent category, ‘pedagogical foundations’. This discussion outlines details of the category’s axial codes, descriptions, properties and paradigm model.

5.1 Open Coding

As previously described, the open coding process consisted of several iterations. This first pass of the data identified 111 open codes (free nodes). This is consistent with other GT studies, which typically generate a large number of codes during the first pass (Kinnunen & Simon, 2010).

5.2 Axial Coding

The axial coding process, the second pass of the data, consisted of several iterations. During these iterations both axial codes and open codes were identified, some were refined and some were new, this is consistent with Strauss and Cobin’s approach documented in the latest version of their methodology (2008). Kinnunen and Simon (2010) also found this to be true when completing the axial coding phase of their project.

Four categories emerged from the axial coding process as follows:

- pedagogical foundations
- teaching practice
- technology adoption
- techno-pedagogical nexus

The ‘pedagogical foundations’ category will now be described in detail. Seven axial codes were used to describe this category. Refer to table 1 for details. The ‘pedagogical foundations’ category describes teachers underpinning ideals, values and philosophy of teaching. This includes thoughts, reflections or comments that demonstrate the thinking behind the practice. This category tells a story of factors and influences which underpin why educators teach the way they do.

Axial Code	Description
Causal factors	The relationship, counsel, guidance, and lessons learned from various teaching role models such as mentors, professional development activities, formal education experiences, conferences etc...
Discipline preference	Identification of learning and teaching techniques tailored to various sub-discipline areas and how that translates into educators own experiences and preferences
Educational language	Examples of educators using educational language to describe practice
Pedagogical development constraints	Perceived obstacles and fears constraining or limiting the development of pedagogical philosophy
Quality teaching attributes	Thoughts, reflections and comments about attributes of quality teachers and what constitutes quality teaching practice
Reflective focus	Describes elements educators reported reflecting on in relation to the development or influence of their teaching philosophy and underpinning values
Understanding of students	Encapsulates educator’s reflections and perceptions and emulation of student learning approaches

Table 1: Axial Codes

Within each, axial code of the ‘pedagogical foundations’ category a number of properties were identified these properties form the characteristics or features that distinguish this category. Refer to table 2 for details.

Axial Codes	Properties
Causal factors	mentor influence

Axial Codes	Properties
	literature professional development
Discipline preference	logic based skills based theoretical
Educational language	learning theory methodology teaching theory
Pedagogical development constraints	industry experience self confidence technology university policy
Quality teaching attributes	communicator empathy entertaining honest passion respect
Reflective focus	student learning teaching knowledge technology use
Understanding of students	engagement motivation learning approach

Table 2: Axial Properties

The paradigm model tool helps to develop a deeper understanding of the categories, and functions as a lens, promoting viewing of categories from different perspectives. For example, instead of thinking about categories, axial codes and dimensions in a hierarchical sense, the paradigm model promotes viewing in a contextual sense. Contextual factors include the conditions, interactions and consequences in which factors influence the pedagogy of tertiary IT educators. The contextualisation factors will aid theory development during phase two of this study. Details of the paradigm model for ‘pedagogical foundations’ is shown in table 3.

Element	Description
<i>Phenomenon:</i>	<i>Pedagogical Foundations</i>
Conditions	Educators reflect on teaching approach in response to publication of allocated teaching load, (particularly for a new course not taught before), when interacting with students, and when using technology to facilitate teaching and learning.
Interactions and Emotions	Educators read literature, attend conferences and discuss concepts with other IT educators.
Consequences	Reflection and change in practice arises in response to educator’s experiences, and interactions.

Table 3: Paradigm Model

6 Discussion

The aim of this section is to provide an analysis and discussion of the ‘pedagogical foundations’ category. Seven axial codes emerged from the data, these codes describe factors influencing the development, growth and formation of tertiary IT educator’s pedagogies and form a

scaffold of support upon which tertiary IT educators reported in this study build their practice.

6.1 Causal Factors

Causal factors describe determining or causal elements or factors. For example education is an important determinant of one’s outlook on life (Farlex Inc., 2011). In terms of this study, causal factors reported include, the influence of and relationships with mentors, engagement with the literature, and participation in professional development activities.

A strong relationship with mentors from early on instils a sense of collegiality and a preference for working with others in team based teaching and learning environments.

“I always tend to work with other people, rather than in isolation, in some ways I have had lots of mentors”

Educators’ own learning experiences from very early on help them to discern between approaches and develop a tool kit of their own learning and teaching techniques.

“I found him fantastic because he just had that really good teaching style”

“very enthusiastic, always available to talk to you about things, even beyond what was actually being taught in the course at the time ... made an impact on me”

Educators found they assimilated valuable techniques, which enhanced their teaching practice through attendance at conferences, professional development activities, observing others, and, participating in peer review sessions.

“I went to a conference and they highlighted the idea of early assessment”

“I watched her give a presentation one day, I watched her pause, and I thought ah yes that’s effective”

Educators indicated they apply research techniques and use the work of others to guide and direct their own practice.

“So I went through and decided, ok, what are the topics that we need to go through. What’s a good order, I looked in text books, and online and I looked at other courses that people had delivered”

“the educationally critical aspects. Somehow they need to be determined. Often it’s by reading the research of other people”

This data provides evidence that causal factors play a role in shaping tertiary IT educator’s pedagogies. In order to maximise the potential of mentoring relationships, it is important to build an environment, which fosters both formal and informal connections between teachers. In addition, educators can benefit from peer review and observation of other teachers in action (see Carbone & Kaasboll, 1998). This observation is supported by the literature, confirming peer reviews provide professional

development opportunities for teachers and a forum to share information about teaching (Marsh, 2008). Educators need ready access to current discipline based educational literature, training programs and support to attend practice based teaching and learning conferences where current theoretical frameworks, tools and techniques are shared.

Possible implications of causal factors: Tertiary IT educators could benefit from access to an environment that fosters relationships with mentors, facilitates access to teaching and learning literature and encourages attendance at professional teaching and learning development activities.

6.2 Discipline Preference

The participants reported three reasons why they were attracted to teaching sub-discipline areas of IT: the theoretical knowledge, the underpinning logic of the content, and having success learning it during their own educational pursuits.

Educators indicated they liked teaching sub-disciplines of IT due to the theoretical content or the logical thinking required.

“I placed a very heavy emphasis on understanding rather than memory work, and, I was interested in how things worked from the theoretical point”

“programming is a completely different way of thinking, it’s very logical”

An underpinning theme reported by educators was the attraction in having been successful in their own studies. The knowledge that they could do it well was an empowering factor for wanting to teach it.

“I think that I felt more on top of the content”

“I mean I was successful at it and I think that was the reason that I decided that I wanted to continue”

This data suggests that educators feel most confident in teaching content that they are familiar with and that suits their own learning approach and interests, whether that is theory based, logic based or skills based within the IT discipline. By enjoying what they are teaching, and feeling confident in their knowledge of the content, tertiary IT educators are more likely to deliver quality teaching and learning outcomes.

Possible implications of discipline preference: Tertiary IT educators feel most confident teaching content they are familiar with and find interesting.

6.3 Educational Language

The participants described a range of teaching frameworks and theories reported in literature but did not appear to be consistent with the language required to connect their descriptions to identifying educational labels. For example, problem-based and applied learning, student-centred learning, constructivism, and learning styles were all described using everyday language. This is

consistent with observations of other researchers (see Harris, 2005).

Educators described attributes of constructivist learning theory without providing the label.

“they go to a lecture, maybe do the homework problems, build up that foundation, build on it for the next portfolio. So it’s a building process”

“I would try and design it so that they could work on small parts each week, and encourage them in the class”

Educators understood the need for applied real world problems, but without providing the labels.

“to get good understanding of the way in which IT is used in the world”

Use of educational language will encourage exploration of the theoretical frameworks underpinning these, leading to a more sophisticated informed approach to solving teaching and learning problems. We encourage this with our students, for example, one participant reported the following about a programming class:

“I had them working on terminology, because I’m a firm believer in that they understand what the terms are and that they can talk about them”

By modelling this behaviour ourselves alongside the expert language of our disciplines, we move into the realm of expert teaching and learning educators.

An environment, which encourages tertiary IT educators’ use of contemporary teaching and learning language will strengthen the use of educational terms in collegial discussions. Use of educational language facilitates a move in educational decision making from a sub-conscious level to a conscious level. It is important to be aware of why we teach the way we do. Intuitive practice is a great base however, by becoming aware, we can access and trial a range of strategies and approaches which can lead to better learning outcomes for students.

Possible implications of educational language: Tertiary IT educators could benefit from using contemporary educational language and conscious decisions to access a range of teaching and learning strategies.

6.4 Pedagogical Development Constraints

Educators reported being constrained in their practice by a number of factors including a lack of industry exposure, lack of self-confidence in front of students during teaching, frustration with technology and infrastructure, and limitations imposed by university policy requirements. These factors worked to undermine their sense of satisfaction and control, and in some cases led to perceptions of unsatisfactory teaching and learning experiences.

A lack of real world commercial experience was reported as a concern. Teachers reported no real world commercial IT experience with which to enrich teaching and learning experiences.

“I have never been involved, apart from minor projects, with the commercial and business side of IT”

“I just think that a well rounded lecturer will have research interests and research experience, will have commercial experience, to get good understanding of the way in which IT is used in the world, as well as a good theoretical knowledge of IT”

Some educators reported a lack of self-confidence and feeling under pressure when teaching students, and claimed that this affected the quality of the students learning.

“when you are out the front of the class you are in a stress situation, and I find when I am doing solutions on the board that sometimes I make mistakes, and I think those mistakes are particularly confusing for the weaker students”

Educators reported a lack of confidence in technology reliability, useability and prior negative experiences when using technology.

“I’m too afraid of everything going wrong. I didn’t buy a CD player until they had been on the market for four or five years, I am a generation behind in my games consoles, I just have never been the person to go out and grab the technology straight away. I let someone else find all the problems first then adopt it”

“I ran into some issues using a multimedia unit a couple of years ago so I tend not to use that anymore”

“with the console, you can actually see the image of what’s on the screen in front of you I tend to stick pretty close to the console”

There was a sense of inflexibility at traditional teacher-centric teaching and learning policies as not accommodating contemporary pedagogy.

“You must have a fifty percent exam and that changes assessment from being formative to being summative, and when it is summative it is too late to fix problems and so I would prefer to have portfolio sessions, mid semester test, and a final test, and have the final test not actually worth very much at all”

This data suggests that there are inhibiting factors, which restrict tertiary IT educator’s ability to enact their preferred pedagogy. Educators need an environment where they can access career long connections with industry. This will enable currency, and embedding of real life experiences. A reliable technology infrastructure is essential to solicit and maintain educator confidence and encourage explorative practice. A move from teacher-centric policy to learner-centred policies will provide educators with flexibility to develop innovative practice. This is supported by Tutty, Sheard and Avram (2008) who found many IT academics are constrained by current government and institution policy resulting in

unsatisfying teaching and learning experiences for both teachers and students.

Possible implication of pedagogical development constraints: Tertiary IT educators’ pedagogies may be inhibited by a lack of confidence in technological infrastructure and traditional university teaching and learning policies.

6.5 Reflective Focus

Teachers reflect on learning that they see occurring in their classes and this shapes their approach in future teaching and learning encounters.

“Last year teaching the same course I found that during tutorials students were often working on their portfolio questions with each other, and that kind of talking is good, so I’m not too sure whether on the alternative weeks whether some sort of group assessment discussion task might be appropriate, that I emphasize assessment is what makes them work”

The opportunity for some continuity in teaching the same or similar courses gives tertiary IT educators time to reflect on their approach so that it can be refined and improved in future iterations of teaching.

Possible implication of reflective focus: Tertiary IT educators may benefit from the opportunity to teach the same or similar content to enable reshaped teaching approaches to be trialled.

6.6 Quality Teaching Attributes

Commonality was observed in the notion of what makes a quality teacher. Educators identified a caring empathetic approach, honesty, enthusiasm, and passion as being the main attributes for great teachers.

“Well they have to be a good communicator, and to different levels, so it can’t just be, being able to, they have to be able to explain things in ways that various different people understands”

“keep it entertaining, so you will engage the students”

“but admitting to something when you are out of your depth”

“demonstrating a passion for my students, and for what I am teaching”

“enthusiasm an absolute must. If the teacher doesn’t seem to be interested in the topic it is very hard to expect the students to be enthused about it either”

“treat students with respect”

In research by Biggs and Moore (1993), these attributes appear as items in the top fifteen functions of great teachers. Biggs and Moore emphasise that these attributes are consistent with the social side of teaching and the connection to students. Given our notion of pedagogy as becoming student centred, it is essential contemporary teaching and learning environments foster

and encourage growth of these attributes, by helping to shape teaching in student focussed way.

Possible implication of quality teaching attributes: Tertiary IT educators could benefit from access to teaching and learning environments (communities of practice) which foster and encourage development of these values.

6.7 Understanding of Students

Educators reported trying to imagine the learning process for students and model their practice around this. In this way educators' pedagogies are influenced by their understanding of students needs.

"I have been more interested in thinking of ways in which I could help students to understand"

"I've always thought about students. Are students going to be able to cope with this? Have they got sufficient support to be do this?"

This data suggests that teachers make assumptions about student learning and mould their practice around these. The difficulty here is getting it correct. Marzano (2007) suggests "A teacher's beliefs about students' chances of success in school influences the teacher's actions with students, which in turn influence student's achievement" (p. 162). Marzano suggests this is perhaps one of the most powerful factors influencing teaching because educators are typically unaware – this is an unconscious activity. It is important then for educators to be encouraged to spend some time reflecting on their approach.

Possible implication of understanding students: Tertiary IT educators could benefit from a conscious awareness of making assumptions about student learning in order to avoid limiting learning options.

7 Conclusion

Current research suggests that educators' understandings of pedagogy have become more complex, and show a move toward technology-enhanced student-centred practices. A variety of influences on pedagogy have been reported in the literature (see section 2.2).

Using a Grounded Theory approach, results from our study show a number of factors that influence how tertiary IT educators develop their pedagogy. Findings from our study suggest that tertiary IT educators think about their teaching and develop their practice in distinct ways, through prioritisation of aspects of teaching and learning that they deem important. One surprising finding was IT educators' approaches to technology-enhanced teaching, in particular their lack of comfort with it. Keeping in mind these findings are from a pilot study of four participants, this techno-pedagogical relationship within a tertiary IT teaching context warrants further investigation.

A holistic approach to encouraging tertiary IT educators to reflect on the factors reported is suggested. By adopting an integrated approach, the key elements can be systematically incorporated in to educational support systems, policy and practice. As tertiary IT educators, we

need to move ourselves from the subconscious doing to the conscious knowing.

8 References

- Bain, K. (2004). *What the best college teachers do*. Cambridge, Massachusetts: Harvard University.
- Beetham, H., & Sharpe, R. (2007). *Rethinking pedagogy for a digital age*. Milton Park, Oxon: Routledge.
- Biggs, J. (2007). *Teaching for quality learning at university: What the student does* (2nd ed.). Maidenhead, Berkshire: Open University Press.
- Biggs, J., & Moore, P. (1993). *The process of learning* (3rd ed.). Melbourne, Vic: Prentice-Hall.
- Canning, J. (2007). Pedagogy as a discipline: Emergence, sustainability and professionalisation. *Teaching in Higher Education*, 12(3), 393-403.
- Carbone, A., & Kaasboll, J. (1998). A survey of methods used to evaluate computer science teaching. In G. Davies & M. ÓHigeartaigh (Eds.), *Proceedings of the Third Annual Conference on Innovation and Technology in Computer Science Education* (pp. 41-45). Dublin, Ireland: ACM.
- Chapuis, L. (2003). *Pedagogy*. ACT: Education and Training.
- Charmaz, K. (2006). *Constructing grounded theory - A practical guide through qualitative analysis*. Trowbridge, Wiltshire: Sage.
- Charmaz, K., & Bryant, A. (Eds.). (2007). *The SAGE handbook of grounded theory*. London, UK: SAGE.
- Conner, M. L., Wright, E., Curry, K., DeVries, L., Zeider, C., Wilmsmeyer, D., et al. (1996). *Learning - The critical technology*. St. Louis, Missouri: Wave Technologies International Inc.
- Crotty, M. (1998). *The foundations of social research - Meaning and perspective in the research process*. London, UK: SAGE.
- Dunican, E. (2006). Initial experiences of using grounded theory research in computer programming education. In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds.), *Proceedings of the 18th Workshop of the Psychology of Programming Interest Group* (pp. 183-197). Sussex: University of Sussex.
- Farlex Inc. (2011). *The free dictionary*. Retrieved August, 26, 2011, from <http://www.thefreedictionary.com>
- Glaser, B. (1978). *Advances in the methodology of grounded theory - Theoretical sensitivity*. Mill Valley, California: Sociology.
- Glaser, B., & Strauss, A. (1967). *The discovery of grounded theory: Strategies for qualitative research*. Piscataway, NJ: Rutgers.
- Goulding, C. (2007). *Grounded theory - A practical guide for management, business and market researchers*. London, England: SAGE.
- Gregory, M., W. (2001). Curriculum, pedagogy, and teacherly ethos. *Pedagogy*, 1(1), 69-89.
- Harris, C. R. (2005). Developing basic online teaching skills, encouraging experimentation. *Distance Education Report*, 9(11), 5-8.

- Hazzan, O., Dubinsky, Y., Eidelman, L., Sakhnini, V., & Teif, M. (2006). Qualitative research in computer science education. In D. Baldwin, P. Tymann, S. Haller & I. Russell (Eds.), *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (pp. 408-412). Houston, Texas, USA: ACM.
- Hoda, R., Noble, J., & Marshall, S. (2010). *Using grounded theory to study the human aspects of software engineering*. Paper presented at the Human Aspects of Software Engineering, Reno, Nevada.
- Jonassen, D., Howland, J., Marra, R., & Crismond, D. (2008). *Meaningful learning with technology* (3rd ed.). Upper Saddle River, New Jersey: Pearson Prentice Hall.
- Kemmis, R., & Smith, E. (2006). *Discipline specific pedagogy*. Retrieved 11 April, 2009, from http://www.icvet.tafensw.edu.au/ezone/year_2006/jul_aug/litreview_discipline.htm
- King, A. (1993). From sage on the stage to guide on the side. *College Teaching*, 41(1), 30-35.
- Kinnunen, P., & Simon, B. (2010). Building theory about computing education phenomena: A discussion of grounded theory. In C. Schulte & J. Suhonen (Eds.), *Proceedings of the Tenth Koli Calling International Conference on Computing Education Research* (pp. 37-42). Koli, Finland: ACM.
- Kollanus, S., & Isomottonen, V. (2008a). Test-driven development in education: Experiences with critical viewpoints. In J. Amillo & C. Laxer (Eds.), *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education* (pp. 124-127). Madrid, Spain: ACM.
- Kollanus, S., & Isomottonen, V. (2008b). Understanding TDD in academic environment: Experiences from two experiments. In A. Pears & L. Malmi (Eds.), *Proceedings of the 8th International Conference on Computing Education Research* (pp. 25-31). Uppsala, Sweden: Uppsala University.
- Kutay, C., & Lister, R. (2006). Up close and pedagogical: Computing academics talk about teaching. In D. Tolhurst & S. Mann (Eds.), *Proceedings of the Eighth Australasian Computing Education Conference* (pp. 125-134). Hobart, Tasmania: ACS.
- Ladwig, J. (2005). Monitoring the quality of pedagogy. *Leading & Managing*, 11(2), 70-83.
- Ladwig, J., & King, M. B. (2003). *Quality teaching in NSW public schools - An annotated bibliography*. Sydney, NSW: Department of Education and Training.
- Lister, R., Berglund, A., Box, I., Cope, C., Pears, A., Avram, C., et al. (2007). Differing ways the computing academics understand teaching. In S. Mann & Simon (Eds.), *Proceedings of the Ninth Australian Computing Education Conference* (pp. 97-106). Ballarat, Victoria: ACS.
- Loughran, J. (Ed.). (1999). *Researching teaching: Methodologies and practices for understanding pedagogy*. Abingdon, Oxon: Routledge Farmer.
- Lynch, J., Sheard, J., Carbone, A., & Collins, F. (2005). Individual and organisational factors influencing academics' decisions to pursue the scholarship of teaching ICT. *Journal of Information Technology Education*, 4(1), 219-236.
- Marsh, C. (2008). *Becoming a teacher - Knowledge, skills and issues*. Frenchs Forest, NSW: Pearson/Prentice Hill.
- Marzano, R. (2007). *The art and science of teaching: A comprehensive framework for effective instruction*. Alexandria, VA: ASCD.
- McNabb, D. (2010). *Research methods for political science: quantitative and qualitative approaches* (2nd ed.). Armonk, New York: M.E. Sharpe.
- Mortimore, P. (Ed.). (1999). *Understanding pedagogy and its impact on learning*. Thousand Oaks, California: Sage.
- Myers, M., D. (1997). Qualitative research in information systems. *MS Quarterly*, 21(2), 241-242.
- Newson, J. (1999). Techno-pedagogy and disappearing context. *Academe*, 85(5), 52-56.
- Olson, D., R., & Bruner, J., S. (1998). The handbook of education and human development. In D. Olson, R & N. Torrance (Eds.), *Folk psychology and folk pedagogy*. Malden, Massachusetts: Blackwell.
- Ramsden, P. (2003). *Learning to teach in higher education* (2nd ed.). Abingdon, Oxon: Routledge Falmer.
- Raths, J. (2001). Teachers' beliefs and teaching beliefs. *Early childhood research and practice*, 3(1), 1-9.
- Schilb, J. (1999). Histories of pedagogy. *College English*, 61(3), 340-346.
- Shulman, L. (2004). *The wisdom of practice: Essays on teaching, learning, and learning to teach*. San Francisco, CA: Wiley.
- Shulman, L. (2005). Pedagogies. *Liberal Education*, 91(2), 18-25.
- Smith, T., & Lowrie, T. (2002). What is pedagogy anyway? *Practically Primary*, 7(3), 6-9.
- Strauss, A., & Corbin, J. (1990). *Basics of qualitative research - Grounded theory procedures and techniques*. Newbury Park, California: Sage.
- Strauss, A., & Corbin, J. (2008). *Basics of qualitative research* (3rd ed.). Thousand Oaks, California: SAGE.
- Tutty, J., Sheard, J., & Avram, C. (2008). Teaching in the current higher education environment: Perceptions of IT academics. *Computer Science Education*, 18(3), 171-185.
- Waters, M. (2005). *Pedagogy in VET - A background paper*. Melbourne, Victoria: William Angliss Institute of TAFE.

Common Areas for Improvement in ICT Units that have Critically Low Student Satisfaction

Angela Carbone, Jason Ceddia

Office of the Pro Vice-Chancellor (Learning and Teaching)

Monash University

PO Box 197, Caulfield East 3145, Victoria

angela.carbone@monash.edu, jason.ceddia@monash.edu

Abstract¹

Unit evaluations across many Australian universities indicate that close to 10% of units in Information and Communication Technology (ICT) disciplines are flagged as needing critical attention. Poor unit evaluation results may lead to a number of negative consequences including poor student learning. To develop an understanding of the reasons why students rate some ICT units as poor, qualitative responses to Monash's unit evaluation questionnaire were examined from 13 ICT units for semester 2, 2010 that were deemed needing critical attention. Responses from students to the question "What aspects of this unit are most in need of improvement?" were analysed. A partial grounded theory based approach was used to code 281 responses to determine common re-occurring themes. Results show eight broad areas in which units can be improved. However the top concern for students in these units is the lecture content. The implications of our results will help ICT lecturers with planning their next unit offering, and will offer some empirical evidence to central teaching preparation programming.

Keywords: ICT Education, education quality in ICT, teaching strategy, ground theory analysis

1 Introduction

There is an increasing amount of attention on the quality of teaching and student satisfaction of units across universities globally. Evaluations of teaching and student experiences within units and courses are now standard practice in Australian universities. Student evaluation of teaching and units, often referred to as SETU, are collected in many institutions.

SETU type instruments are usually administered towards the end of each semester, and results are analysed to provide a "snapshot" of students' perceptions of their teachers, the unit and their learning. Most data is gathered via simple to use and administer

unit evaluation questionnaires. Students rate a unit on a Likert scale, followed by opened ended questions in which they can state the best aspects of the unit and areas for improvement.

The nature and composition of unit evaluation instruments used by tertiary institutions across Australia offering ICT degrees were recently discussed at a workshop on unit evaluation practices, at the 2011 Australia Council of Deans of Information Communication Technology (ACD ICT), Learning and Teaching Forum (Australia Council of Deans of Information Communication Technology (ACD ICT), 2011). Unit evaluation instruments from fourteen universities, across five Australian states (Victoria, NSW, Queensland, Tasmania and South Australia) were reviewed by 24 workshop attendees. Attendees mainly comprised Deans, Associate Deans Education (ADE's), Heads of schools, and academics in leading education roles.

In a workshop activity, the nature and composition of unit evaluation instruments was discussed along with the educational intentions of those who use the data from these instruments. Across all institutions the nature of the instrument was broken into a set of closed questions followed by a small number, usually two or three, open ended questions, with one of the open ended questions asking students to identify what aspects of the unit need improvement.

Participants also agreed that many stakeholders use the unit evaluation results in the following ways: academics use the results to make practical improvements to their units and course; some Heads of School use the results to determine an academic's funding allocation, and to identify the type of support they can offer their staff; Deans and ADEs determine whether educational targets are met; Faculty Education Committees identify whether their faculty is performing well, and University Education Committees use the results to compare performance of faculties against other faculties within the university and to determine whether educational targets have been met.

Unfortunately, unit evaluations across many Australian universities indicate that close to 10% of units in ICT disciplines are flagged as needing critical attention (Australian Graduate Survey - Course Experience Questionnaire (CEQ) 2005-2009). Low performing units can affect student learning, have a negative effect on the morale of the lecturer, place pressure on Deans and ADEs to improve educational performance and can cause universities to fail to meet national targets on educational performance which may

¹ Copyright © 2012, Australian Computer Society, Inc. This paper appeared at the Fourteenth Australasian Computing Education Conference (ACE 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 123. M. de Raadt and A. Carbone, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

result in reduced government funding to universities. Consequently, ICT faculties often struggle to meet university and national targets on educational performance.

In order to help academics improve units in need of critical attention a Peer Assisted Teaching Scheme (PATS) was developed (PATS, 2010). PATS was originally piloted in the Faculty of Information Technology (FIT) in 2008 at Monash University, Caulfield Campus, Australia (Carbone, 2011b). The scheme builds upon peer facilitation and how it contributes to the development of academics (Ashwin, 2003) and developing mentoring relationships (Gratch, 1998). Following its initial success, PATS was funded by a 2010 ALTC Teaching Fellowship (Carbone, 2010) so that now PATS is open to all academics who wish to improve the health and quality of their units. The scheme aims to inform and equip academics with skills and strategies to improve their units and build peer capacity to enhance learning and teaching (Carbone, 2011a, Ashwin, 2003, Gratch, 1998)

The challenge though, is to develop an understanding of why some ICT units are rated low on unit evaluations and to build a picture of the 'critical issues' or areas that need improvement, so that these are at the forefront of academics' minds when planning, implementing and delivering units.

This paper reports on a partial grounded theory approach to analyse the SETU unit evaluation qualitative comments in the ICT discipline for unit that students perceive as needing critical attention at Monash University.

2 Background

At the 2011 ACD ICT Learning and Teaching Forum, it was evident that most ICT faculties administered some form of SETU instrument. This widely used instrument usually contains two components: the teaching evaluation instrument, and the unit evaluation instrument. In this study we focus on the responses to the unit evaluation instrument at Monash University.

The unit evaluation component of SETU focuses on student perceptions of units. These surveys are extremely important in identifying units that are meeting students' expectations and needs, as well as units that can be improved.

2.1 Monash Unit Evaluation Instruments

Like most universities, Monash University distributes SETU surveys at the end of each semester. In the Faculty of IT all units are evaluated every semester using an online survey. There are five university wide (UW) unit evaluation items. These are:

- UW-Item 1 The unit enabled me to achieve its learning objectives
- UW-Item 2 I found the unit to be intellectually stimulating
- UW-Item 3 The learning resources in this unit supported my studies
- UW-Item 4 The feedback I received in this unit was helpful

UW-Item 5 Overall I was satisfied with the quality of this unit

Responses to these questions use a five point Likert scale ranging from Strongly Agree (5) to Strongly Disagree (1) and with 3 representing "Neutral". Options for Not Applicable (6) and Don't Know (7) are also provided to respondents and are not counted in the means for questions.

Reports generated from the analysis of the closed question responses for all units are publicly accessible by Monash staff and students (Monash University, 2011b). One of the key measures used in the reports is the 'median'. The median is calculated under the assumption that the five point scale represents a continuous random variable rather than five discrete categories.

Immediately following the closed questions, there are two open-ended questions:

1. What were the best aspects of this unit?
2. What aspects of this unit are most in need of improvement?

Only academic staff, and their superiors have access to these comments.

2.2 Monash Unit Quality Indicators

Monash University focuses on item 5 (reporting overall satisfaction) in providing university managers with a quick way of monitoring aggregate performance of the unit. Using item 5 as the key question, a "traffic light" indicator was then developed to interpret the results.

Any unit with a median value of 3.0 or below to the UW-Item 5 "Overall I am satisfied with the quality of the unit" is flagged as needing critical attention. Any unit between 3.01 and 3.59 indicates that the unit needs improvement because responses are generally "neutral" or bimodal with no clear trend. Any unit between 3.6 and 4.69 indicates that the unit is meeting aspirations because responses are generally above "neutral", and the great majority are "agree" or "strongly agree". Any unit scoring above 4.7 indicates that the majority of responses are in strong agreement that the unit is outstanding. Table 1 summarises the meaning of the unit quality indicators.

Colour Code	Meaning	Unit Measure	Characteristics of unit response
Purple	Outstanding	Median \geq 4.7	Majority of responses are "strongly agree"
Green	Meeting aspirations	Median between 3.6 – 4.69	Responses are generally above "neutral", the great majority are "agree" or "strongly agree"
Orange	Need to improve	Median between 3.01 – 3.59	Responses are generally "neutral" or bimodal with no clear trend
Red	Needing critical attention	Median \leq 3.0	Responses generally below "neutral", majority "disagree" or "strongly disagree"

Table 1: Monash unit quality indicators

The target set by Monash University is that 5% or more units should be rated as “outstanding”, 80% or more should “meet aspirations”, 10% or less should “need improvement” and 5% or less should “need critical attention”. At the end of each semester a “red report” is produced flagging units that fall in the needing critical attention zone. For these units, the academic policy on Student Evaluation of Teaching and Units (SETU) (Monash University, 2011c) Procedures requires that:

“Each unit-owning faculty reviews the published reports and data files of the unit evaluation data and prepares an action plan to address areas for improvement for faculty-wide issues.”

and that

“The department/school prepares an action plan to address areas for improvement where unit issues are identified.”

Units that fall in the “red” for three consecutive offerings are deemed non viable and are discontinued, unless the Dean or ADE argue a case for their continuation along with a detailed action plan.

Monash University has set a target of less than 5% for units requiring critical attention. Unfortunately, figures from 2008 to 2010 ICT unit evaluation surveys show that approximately 10% of units within ICT need urgent attention (Monash University, 2011d).

3 Research Approach

This section describes how the responses to the open ended questions in the unit evaluation data was obtained, and details the process followed to analyse the data. Since the raw data was not collected by the project team, the team sought permission to use the data gathered by University Statistics (Strategic Analysis and Surveys), from of the Office of Pro Vice-Chancellor (Planning & Quality), (OPVCPQ).

Human ethics approval was obtained to analyse the unit evaluation qualitative comments for the units needing critical attention before commencement of the project.

3.1 Unit vs. Unique Unit Offering

To obtain the data from the OPVCPQ a clarification was required about the term unit. FIT teaches its units across multiple campuses. Monash has six campuses, four domestic campuses within Victoria and two international campuses, Malaysia and South Africa.

This essentially means that the same unit can be offered at six different campuses. In a Unit Evaluation, a 'unit' is defined in a slightly different way; it is a 'unique unit offering', which is a unique identifier comprising the following components:

unique unit offering = unit code + teaching period + mode (eg. face-to-face or off-campus) + location (eg. campus)

For example, as shown in Table 2, the fictitious unit FIT1234, may have four unique unit offerings, with

different overall satisfaction ratings across the different campuses, some of which may be above 3, and some below. “Mode” in Table 2 refers to the delivery mode; ‘f2f’ refers to face-to-face and OCL refers to ‘Off Campus Learning’.

Unit Code	Sem	Year	Mode	Campus	UW-Item 5
FIT1234	2	2010	f-2-f	campus-A	3
FIT1234	2	2010	OCL	campus-B	2.7
FIT1234	2	2010	f-2-f	campus-C	4.7
FIT1234	2	2010	mixed	campus-D	3.9

Table 2: The same unit offered at four campuses

The average median for all the unique unit offerings is 3.57, which is well above 3. However, there are two unique unit offering (FIT1234 campus-A and campus-B) with median 3 or below. For this study, the qualitative comments for all the unique unit offerings of ICT units that were taught in semester 2, 2010 that scored 3.0 or below were requested.

3.2 The Data Collection

The OPVCPQ extracted the comments from the 'unique unit offerings' with median of 3 or below for all faculties. Comments relating to the same unit were consolidated into one file and put into a folder of the unit owning faculty. In the above example, the comments of FIT1234 as surveyed at campus A and Campus B were put together in one file and stored in the folder of 'ICT', though they are treated as two unique unit offerings.

Ten faculty folders containing comments from various unique unit offerings with the same 'unit code' were provided. However, for this study only student comments relating to the ICT units were analysed.

The OPVCPQ provided comment files with campus and unit information removed from the files. The majority of the comments that were provided came from online surveys, however, a small portion of the hand written comments taken from the paper surveys where provided as images. Some 'unique unit offerings' had no comments at all. The comments in the provided files were partially de-identified, with unit and campus information being removed. However, some files contained students' comments with sensitive information that could possibly lead to the identification of staff, so all identifying information was removed before using these comments in this study.

3.3 Method of Analysis

Once the data was converted into a non-identifiable form a consistent analysis approach was needed. Two methods of analysis were considered. These were:

- i) Using a similar approach to that reported in the Course Experience Questionnaire comments summary report by Monash Quality Unit, where a count of the positive to negative comments is undertaken and then a ratio calculated (Monash University, 2011a). The ratio would provide an indication of 'balance of opinion' but not the areas in which students saw as needing critical attention.

- ii) Using a grounded theory based approach to code the data to determine common re-occurring themes in need of critical attention (Dick, 2005, Strauss and Corbin, (1990).)

Since the aim is to develop an understanding of common areas for improvement in units needing critical attention, the grounded approach (ii) was adopted. However, only open coding was performed as this is still an exploratory study and so no causal connections have been postulated between categories i.e. no axial coding has been done. Thirteen ICT units were identified by OPVCPQ as needing critical attention and these units were referred to as unit 1 to unit 13. The student comments to the following open ended question were analysed:

“What aspects of this unit are most in need of improvement?”

The main categories were arrived at by the two researchers independently reading through all the comments for the unit with the most comments and listing common themes. The main categories were straightforward to identify as they were effectively ‘keywords’ in the comment. For example, a comment may begin with “The lecturer was...” indicating that this comment is in the ‘lecturer’ category. This process was repeated for a further two units, those with the second and third most comments. Comparison of coding showed little disagreement amongst the researchers. The category/sub category comment frequency per unit as well as the number of unique unit offerings for the unit is shown in Table 3; the top three units with the most comments were units 2, 1 and 11.

Unit	Category Comment Frequency	Number of unique offerings
2	62	5
1	50	6
11	35	1
13	33	2
5	28	3
8	22	2
9	13	3
12	12	1
7	9	1
4	7	2
3	6	2
10	2	2
6	1	1

Table 3: Comments per unit in decreasing frequency order

For each category, the researchers identified a set of category attributes. Arriving at a common set of attributes required considerable negotiation between the researchers. Once the main categories and the category attributes were agreed, the researchers divided the remaining units and each coded the units separately.

While there was scope to add categories and attributes should they appear in the subsequent coding of comments, this did not occur.

During the final phase of coding, each researcher checked a random set of codes from the other researcher, and discrepancies were discussed; of the 25 checked only 2 were discussed giving an error rate of 8% which was deemed acceptable.

This coding exercise has similarities to the World Health Organisation (WHO) coding the cause of death from pathologist’s reports. WHO uses a coding system called the International Classification of Diseases (10th revision) abbreviated to ICD10 (World Health Organisation, 2011). Coders are required to undergo a training course on using the coding scheme to ensure consistency between coders.

To avoid breaching ethical requirements in our reporting we could not report on the total unit enrolment in Table 3. By specifying the unit enrolment and also knowing that the unit scored below 3.0 on UW-Item 5, the year it was taught and the faculty it belonged to, it would be possible for in-house academics and students to identify the unit. We have therefore expressed this concern as a limitation of our study in section 4.5.

4 Results and Discussion

A total of 281 qualitative comments from thirteen ICT units were categorised and coded. The actual number of individual students giving feedback is less than 281 as some students commented on multiple areas and these were coded as separate comments into their respective areas.

4.1 Note About the Units

Of the thirteen units analysed, Table 3 shows that five units (units 6, 7, 10, 11, 12) were delivered as single unit offerings while the others were delivered as multiple unit offerings. The extreme cases were unit 1 being delivered as 6 offerings simultaneously and unit 2 as 5 offerings. No conclusions will be presented for an individual offering as the data has been de-identified so it is not possible to say which comments relate to which offering. Also, different staff were involved in unit delivery at the different locations and again no comment can be made about specific staff.

4.2 The Main Categories

There were eight main categories that emerged from the analysis process. Table 4 contains the eight categories and the number of comments recorded against that category. The percentage figure is the frequency as a fraction of the 281 comments made.

The ‘lecturer’ and ‘lecture’ categories differ in that ‘lecturer’ relates to items like the presentation style, apparent knowledge of the subject matter in answering audience questions and availability to students. ‘Lecture’ refers to the content of the actual lecture as gauged by how much material was presented, the logical flow to the material and the originality of the material.

Likewise the ‘tutor’ and ‘tutorial’ categories differ in that ‘tutor’ relates to how prepared and knowledgeable the tutor was and how responsive to students were they in terms of answering questions and emails. ‘Tutorial’ refers the relevance or alignment of the material to the lecture, the type of exercises, the complexity of exercises and the duration of the tutorial.

Category	Frequency	%
Lecturer	48	17.1
Lecture	80	28.4
Tutorial	55	19.6
Assessment	53	18.9
Tutor	14	5.0
Off Campus	7	2.5
LMS	15	5.3
Resources	9	3.2

Table 4: Main categories

The ‘assessment’ category refers to items like clarity of the assignment specification, alignment with lectures, detailed and clear marking guidelines and quality of feedback. The ‘LMS’ (Learning Management System) category refers to items like ease of navigation, amount of material and accuracy of the material. The ‘Off campus’ category refers to the level of support specifically for off campus students. This may be via the LMS or availability of lecturers and tutors for consultation. The ‘resources’ category refers to the currency of recommended readings, the availability of readings and references from the library and the sheer quantity of readings and references.

4.3 The Attributes of the Main Categories

Each of the main categories contained a set of sub-categories or attributes. Tables 5 to 12 contain the list of attributes from each main category, the total number of comments for each attribute, and percentage expressed as a fraction of the category comments.

Category Attribute	Frequency	%
Lecturer-knowledge	4	8.3
Lecturer-presentation style/engagement	20	41.7
Lecturer-support (availability, attitude)	13	27.1
Lecturer-organisation	10	20.8
Lecturer-response time	1	2.1
Lecturer	48	100%

Table 5: Lecturer attributes – 48 comments

Under the main category of *lecturer*, five attributes emerged from the students’ comments. These were the lecturer’s knowledge, presentation style, the support provided, organisation and response time. In this category, overwhelmingly the lecturers’ presentation style was most frequently mentioned. Typical responses from students about the presentation style included:

- *The lectures were incredibly dull and presented poorly.*

- *THE TEACHING! We just sit in class without any proper guidelines. They expect us to learn from somewhere and just come in and do exercises.*
- *Needs more engaging teaching methods.*
- *I believe that the lecturer’s delivery could use some improvement. It’s just the delivery of his lectures tends to drone.*

Category Attribute	Frequency	%
Lecture-structure	15	18.7
Lecture-access	2	2.5
Lecture-content	51	63.7
Lecture-challenge	5	6.3
Lecture-quantity	7	8.8
Lecture	80	100%

Table 6: Lecture attributes – 80 comments

Under the main category of *lecture*, five attributes emerged from the students’ comments. These were the lecture structure, access to the material, the content, the level of challenge, the amount of material and the accuracy of the material. Overwhelmingly the lecture content was most frequently mentioned. Typical responses from students about the lecture content included:

- *The overall content of the course was very "ideal situation" theory and not real world practicalities.*

Many students have mentioned they find the content superficial and not aimed at “the right” level.

- *The lecture’s content should be more detail and more reading lists suggested*
- *The content seems to be outdated.*
- *The content of this unit should be altered for students to be able to see the relevance of the information given in the REAL world ie how to apply the information in the real world.*
- *Lectures should not refer to SAP screens, especially when using text-only (no screen shots) to explain navigation of SAP interface.*

Category Attribute	Frequency	%
Tutorial-type of activity	12	21.8
Tutorial-clarity	7	12.7
Tutorial-alignment	16	29.1
Tutorial-available software	16	29.1
Tutorial-length	1	1.8
Tutorial-scheduling	3	5.5
Tutorial	55	100%

Table 7: Tutorial attributes – 55 comments

Under the main category of *tutorial*, six attributes emerged from the students’ comments. These were the type of activity, the clarity of the tutorial questions, alignment of the tutorial activity with the lecture content, the available of software to complete the activity, the length of the activity and when the tutorial class was scheduled. The tutorial alignment and availability of software were most frequently

mentioned. Typical responses from students about the tutorial alignment included:

- *Unbelievable amount of incoherence between all elements of the subject-lecturers, tutorial and assignments.*
- *Tutorials not too directly related to what is covered in class or in the book.*
- *tutorials/structure of work is completely unrelated to weekly classes.*
- *Overall structure and ensuring the work in tutorials is relevant to the exam.*
- *Tutorials should be aimed towards learning objectives.*

Category Attribute	Frequency	%
Assessment-marking (consistency of marking, quality of feedback, timeliness, clarity of marking criteria)	20	37.7
Assessment-alignment	8	15.1
Assessment-specification	25	47.2
Assessment	53	100%

Table 8: Assessment attributes – 53 comments

Under the main category of *assessment*, three attributes emerged from the students’ comments. These were related to the consistency of marking, the alignment of the assessment to the learning objectives and the clarity in the assessment specification. In this category, the specification was most frequently mentioned followed by the marking criteria. Typical responses from students about the assessment specification and marking included:

- *The assessments were in great need of updating - outdated directions for use of software that had changed. Non-standardized submission formats that made assessments a frustration.*
- *Clarity of the assessment tasks and assignments*
- *the marking system in this unit is very disappointing and the feedback is terrible. For most assignments; they have not even stated what is done wrong, but just given a grade!*

Category Attribute	Frequency	%
Tutor-knowledge	1	7.2
Tutor-presentation style	2	14.2
Tutor-support	10	71.4
Tutor-response time	1	7.2
Tutor	14	100%

Table 9: Tutor attributes – 14 comments

Under the main category of *tutor*, four attributes emerged from the students’ comments. These were related to the tutor’s knowledge of the subject material (or preparation before the tutorial), presentation style to the students (discussion of questions or just giving answers), how supportive or helpful the tutors were to students and how long did they take before replying to questions. Tutor support was of the most concern to students. Typical comments included:

- *TEACHER SUPPORT! I have not spoken to ANY teacher or tutor in my entire time with this subject. I don't even know if I have a tutor let alone their names. The level of help offered was slim to none for me,*
- *In a Two hour tutorial for this unit. My tutor would spend the first hour checking emails and 'surfing' the web while we completed the tutorial exercises (he did not offer any help or guidance).*
- *we would spend 20 minutes going through the answers where the tutor read off the answer sheet he had and could not provide any greater understanding.. Then the tutor left after 1 hour and 20 minutes EVERY WEEK..*

Category Attribute	Frequency	%
Off-Campus-support	6	85.7
Off-Campus-availability recordings	1	14.3
Off-campus	7	100%

Table 10: Off-campus attributes – 7 comments

Under the main category of *off-campus*, two attributes emerged from the students’ comments. These were related to the amount of support for distance education students and availability of resources that captured what occurred in the lecture. Typical comments included:

- *The off campus lecturer was unavailable to answer questions -repeated posts on Blackboard were ignored. On one occasion, after posting the same question twice and waiting almost two weeks for an answer*
- *There needed to be a recorded lecture made available to students (especially for off campus students). Sometimes, reading something does not achieve the same effect compared to hearing someone talk about it, or describe it more in detail.*

Category Attribute	Frequency	%
LMS-ease of use	4	26.7
LMS-quantity	1	6.7
LMS-accuracy	10	66.6
LMS	15	100%

Table 11: LMS attributes – 15 comments

Under the main category of *LMS*, three attributes emerged from the students’ comments. These were related to the ease of navigation and finding materials (comparing Blackboard to Moodle), the amount of material that was actually on the LMS and accuracy of the material. LMS-Accuracy was the biggest concern as students rely on this information if they missed lectures.

- *The content of the course is a mess. There are old files from previous semesters sitting in the Blackboard which makes it very confusing.*
- *Use of blackboard was poor for this unit. Lecture notes on blackboard were frequently not the ones used in the lecture. When asked for updated notes,*

these were emailed to individuals rather than put up on Blackboard.

The *resources* category dealt with issues like how many readings were there for the unit as well as how available they were in the library or other sources.

Category Attribute	Frequency	%
Resources-relevance	5	55.6
Resources-Quantity	1	11.1
Resources-Availability	3	33.3
Resources	8	100%

Table 12: Resource attributes – 9 comments

Typical comments included:

- *Readings clearly not considered this semester - with the library list it was difficult to determine which was intended for which week, with lecture notes "key readings" some were just unable to be sourced..*
- *Disappointed that most of the reading material for a multimedia unit was ~ 10 years old..*

4.4 The Most Frequently Mentioned Categories

Table 13 lists the most frequently occurring areas of concern to the students. These areas have been aggregated across the 13 ICT units. So, across all 13 units, 51 of the 281 total comments related to *lecture-content*. Clearly this is the main concern for ICT students, followed by *assessment-specification*, and then two similarly ranked items: *lecturer-presentation style* and *assessment-marking*.

Rank	Frequency	%	Category Description
1	51	18.2	Lecture-content
2	25	8.9	Assessment-specification
3	20	7.1	Lecturer-presentation style/engagement
3	20	7.1	Assessment-marking
4	16	5.7	Tutorial-alignment
4	16	5.7	Tutorial-available resources
5	15	5.4	Lecture-structure
6	13	4.6	Lecturer-support
7	12	4.3	Tutorial-type of activity
8	10	3.6	Lecturer-organisation
8	10	3.6	Tutor-support
8	10	3.6	LMS-Accuracy
9	8	2.9	Assessment-alignment
10	7	2.5	Lecture-quantity
10	7	2.5	Tutorial-clarity

Table 13: Top ten category attributes

Although, not all areas listed in Table 13 applied to every unit, many of these areas were raised by students in 2008 and 2009 “red” units. The outcome of this study confirms the preliminary findings already

obtained from earlier PATS participants that are reported in (Carbone, 2011b). Participants in the PATS program typically focused on and addressed one of those areas related to improving the lecturer content, linking the lecture to the tutorial material and improving the clarity of assessment items (Carbone, 2011b).

4.5 Limitations of Study

When interpreting the data it should be noted that units which had a large student cohort could distort perceptions across the faculty. Table 13 shows that lecture content is the biggest concern for ICT units. However, this finding is heavily influenced by unit 2 which had the most comments (62), as listed in Table 3. Table 14 shows the top four concerns for unit 2 as lecture content, lecturer presentation, tutorial alignment and assessment alignment. In tables 14, 15 and 16, the column “% of category” indicates how much this unit influenced the category attribute overall. For example, in Table 14, fourteen comments from a total of 51 comments related to lecture content (ie. 27.5%) come from Unit 2; 10 comments from a total of 20 comments related to lecturer presentation (ie. 50%) also come from Unit 2.

Category description	Unit 2 category frequency	Overall category frequency	% of category
Lecture-content	14	51	27.5
Lecturer-presentation style/engage	10	20	50.0
Tutorial-alignment	7	16	43.7
Assessment-alignment	4	8	50.0

Table 14: Top 4 category attributes for unit 2

Table 15 has the top four concerns for unit 1, which has the second highest number of comments. For unit 1, assignment specification is the top concern, whereas lecture content is the fourth concern.

Category description	Unit 1 category frequency	Overall category frequency	% of category
Assessment-specification	10	25	40.0
Tutorial-available software	9	16	56.3
Lecturer-presentation style/engagement	6	20	30.0
Lecture-content	4	51	7.8

Table 15: Top 4 category attributes for unit 1

However, for unit 8, with the sixth most number of comments and hence chosen as a ‘more average’ unit, resource relevance is of the most concern.

Category description	Unit 8 category frequency	Overall category frequency	% of category
Resources-relevance	4	6	66.7
Lecture-content	3	51	5.8
Lecture-structure	2	15	13.3
Tutorial-alignment	2	16	12.5

Table 16: Top 4 category attributes for unit 8

Hence, while the top concerns are listed in Table 13, it should be noted that they do not necessarily apply to all units and certainly not in that order.

Another limitation of the study relates to response rates. As explained in section 2.1, the Monash SETU instrument has two components; 5 Likert scale (quantitative) questions and 2 open ended (qualitative) questions. A student response is included in calculating the response rate even if the student only responds to the University Items 1-5 (quantitative questions) and leaves the qualitative questions blank. For example, in unit 11, there were 72 enrolments with 22 responses giving a response rate of 30.6%. However, there were only 13 actual qualitative comments, giving a response rate of 18.1%. As shown in Table 3, these 13 qualitative comments gave rise to 35 category/sub category comments. At this stage, the breakdown for unit 11 has only been possible because it had only one unique unit offering. Obtaining these statistics is an area for further investigation, without breaching ethical considerations.

5 Conclusion and Future Work

Close to 10% of units in Information and Communication Technology (ICT) disciplines across Australia are flagged as needing critical attention. Such results may lead to a number of negative consequences including poor student learning. This study aimed to take a first step in developing an understanding of the areas that students perceive as needing critical attention. This understanding was achieved by analysing the qualitative responses to Monash's unit evaluation questionnaire that were deemed needing critical attention.

A partial grounded theory based approach was used to code 281 responses from across 13 ICT units for semester 2, 2010 to the question "*What aspects of this unit are most in need of improvement?*". The 13 ICT units selected were those in the "red zone" as measured by the Monash unit quality indicators in Table 1. Responses from students were used to determine common re-occurring themes, which covered eight broad areas in which units can be improved. These related to the lecturer, the lecture, the tutor, the tutorial, assessment, the LMS, off-campus support and resources. Students identified aspects of units most need of improvement. The units examined by the authors were ones that the students' perceived lacked quality. So by grouping these aspects into categories, the authors have identified eight categories which impact on students' perception of 'unit quality'.

However, the three top concerns for students in these units are the *lecture content*, the *assignment specification* and equal third, the *lecturer's ability to engage students* and the *assessment marking* which includes consistency of marking, quality of feedback, timeliness, and clarity of marking. The implications of our results will help ICT lecturers with planning their next unit offering, and will offer some empirical evidence to central teaching preparation programming. This understanding will inform the design and development of professional staff training programs, to improve units, teaching practice, the student experience and unit evaluations.

The next phase of project is repeat the qualitative comment analysis process described above with unit evaluation data from the remaining faculties. This will be tackled by initially analysing the data from low performing units in the Faculty of Engineering and Faculty of Education, since these faculties generally perform below the university average at Monash University. This process will be followed by a further analysis on the data derived from faculties who generally are top performers (ie. the Faculty of Law, the Faculty of Business and Economics and the Faculty of Arts).

In semester 2, 2010, the reported number of poorly performing units were:

- 19 for Art and Design;
- 33 for Arts
- 20 for Business and Economics
- 29 for Education
- 9 for Engineering
- 1 for Law;
- 37 for Medicine, Nursing and Health Sciences
- 2 for Pharmacy and
- 9 for Science

It is expected that the main eight categories will still be applicable as they are generic to educational concerns across any discipline but that the attributes may change for each category. Of interest is whether the priorities of the categories and attributes remain the same as those identified in the ICT units.

It is also expected that as more faculty units are analysed that some axial coding of the categories should emerge. For example, if the assessment specification is poor then assignment marking may also be a concern.

6 Acknowledgements

The authors wish to thank the Australian Learning and Teaching Council (ALTC) Teaching Fellowship Program for funding an extension grant to the Peer Assisted Teaching Scheme (PATS) to analyse qualitative SETU data for units perceived as needing critical attention across all faculties of Monash University. A special thank-you goes to Ms Jessica Wong for her administrative assistance throughout the project.

7 References

- Ashwin, P. 2003. 'Peer facilitation and how it contributes to the development'. *Research in Post-Compulsory Education*, 8, 5-18.
- Australia Council of Deans of Information Communication Technology (Acad Ict) 2011. Learning and Teaching Forum, University of Adelaide.
- Australian Graduate Survey - Course Experience Questionnaire (Ceq). *National CEQ data - Survey Items Results: 2005-2009* [Online]. Available: <http://www.opq.monash.edu.au/us/pivot-table/> [Accessed Jan 2011].
- Carbone, A. 2010. Peer Assisted Teaching Scheme (PATS). Australian Learning and Teaching Council (ALTC).
- Carbone, A. 2011a. Building peer assistance capacity in faculties to improve student satisfaction of units. *Higher Education Research and Development Society of Australasia (HERDSA)*. Gold Coast, Queensland, Australia.
- Carbone, A., Ceddia J. And Wong J. 2011b. A Scheme for Improving ICT Units with Critically Low Student Satisfaction. *Innovation and Technology in Computer Science Education (ITiCSE)*. Darmstadt, Germany.
- Dick, B. 2005. *Grounded theory - A thumbnail sketch*. [Online]. Available: <http://www.scu.edu.au/schools/gcm/ar/arp/grounded.html>. [Accessed July 26 2011].
- Gratch, A. 1998. Beginning Teacher and Mentor Relationships. *Journal of Teacher Education*, 49.
- Monash University. 2011a. *CEQ reporting data* [Online]. Available: <http://www.opq.monash.edu.au/us/surveys/ags/ceq-comments-analysis/index.html> [Accessed August 2011].
- Monash University. 2011b. *SETU survey results* [Online]. Available: <http://opq.monash.edu.au/us/surveys/unit-evaluations/distribution-administration.html> [Accessed August 2011].
- Monash University. 2011c. *Student Evaluation of Teaching and Units (SETU) Procedures* [Online]. Available: <http://www.policy.monash.edu/policy-bank/academic/education/quality/student-evaluation-of-teaching-and-units-procedures.html> [Accessed August 2011].
- Monash University. 2011d. *Unit evaluation reports* [Online]. Available: https://emuapps.monash.edu.au/unitevaluations/wr/u_evr_rp1_public_yearseme.jsp [Accessed August 2011].
- Pats. 2010. *Peer Assisted Teaching Scheme Resources* [Online]. Available: <http://opvclt.monash.edu.au/educational-excellence/peerassistedteachingscheme/resources.html> [Accessed August 2011].
- Strauss, A. & Corbin, J. (1990). . *Basics of qualitative research - Grounded theory procedures and techniques.*, Newbury Park, California:, Sage.
- World Health Organisation. 2011. *International Classification of Diseases* [Online]. Available: <http://www.who.int/classifications/icd/en/> [Accessed August 2011].

Directions and Dimensions in Managing Cheating and Plagiarism of IT Students

Judy Sheard

Faculty of Information Technology
Monash University
PO Box 197, Caulfield East 3145, Victoria
judy.sheard@monash.edu

Martin Dick

School of Business Information Technology
and Logistics
RMIT University
GPO Box 2476, Melbourne 3001, Victoria
martin.dick@rmit.edu.au

Abstract

The problem of cheating at university is a widespread and long-standing issue. There are a variety of strategies that are used to address the problem which broadly fall into the areas of education, prevention, detection and consequence. An important consideration when deciding to tackle the problem of cheating is that the effectiveness of methods for addressing cheating are not necessarily the same for the different types of cheating. This paper presents an investigation of cheating practice of undergraduate IT students using a factor analysis to determine categories of cheating behaviour and influences on this behaviour. The implications arising from this analysis for addressing cheating are then examined and recommendations made for strategies which are appropriate for particular types of cheating.

Keywords: cheating, plagiarism, undergraduate students.

1 Introduction

The problem of cheating is a difficult one to address as in many ways it is a negative-sum activity for academics. Time and effort devoted to preventing, detecting and punishing students who have been cheating is time that is not being used to teach and to improve the learning experience for students. It is important therefore for academics to determine a balance in their educational activities so as to reduce the incidence of cheating in their classes, but without distorting or at least minimally distorting the educational experience that is being provided to the students.

In order to achieve this aim, it is necessary to consider a range of issues. Previous work by the authors (Dick et al., 2003; Dick, Sheard, & Hasen, 2008) proposed that activities for managing cheating fall into four main conceptual areas:

- **Education** – primarily focused on activities that educate students on what cheating is and why it is bad for them to cheat, but also educating academics on the issue of cheating and how to address it in their teaching
- **Prevention** – activities which assist in the design of curriculum and assessment tasks so as to minimise the need for, and the benefit to be gained from, a variety of cheating types, and also providing tools for students to manage their studies
- **Detection** – the development of tools and procedures to detect when cheating has taken place and the processes by which these tools are used in teaching
- **Consequence** – the procedures in place to handle cases of cheating, the punishments which are allocated for cheating and to publicise the results of those procedures

The academic needs to make decisions on the level of effort that they allocate to the above four areas. In doing so, the academic has to consider the type of cheating that may occur in their classes. The effectiveness of strategies for addressing cheating are not necessarily the same for the different types of cheating. For example, students are often confused about plagiarism (McCabe, 2005) and strategies which focus on education will probably be the most effective to address this issue; however, education-focused strategies may be quite ineffective in addressing situations where students arrange for other people to sit their exam for them. In that case, a focus on consequence and the potential punishments may be the best strategy to adopt.

This paper builds on the results of a survey undertaken at an Australian University (Sheard & Dick, 2011) by using a factor analysis to determine the commonalities in three areas of the survey: the attitudes towards different cheating practices, the reasons that students indicate would cause them to cheat and the reasons that students indicate would cause them **not** to cheat. These three sets of factors were then analysed to determine whether there are statistically significance differences in influences on different types of cheating practices between the students who claimed to have cheated and those who claimed not to have cheated. The focus of the analysis is on the undergraduate student cohort of the dataset. A similar

Copyright © 2012, Australian Computer Society, Inc. This paper appeared at the 14th Australasian Computing Education Conference (ACE2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology, Vol. 123. M. de Raadt and A. Carbone, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

analysis on a postgraduate cohort was conducted by the authors in a previous survey (Sheard & Dick, 2003).

The implications arising from the analysis for strategies and focus to be taken to address particular types of cheating practice are then examined.

2 Background

In order to set the scene for the analysis, this section looks at the extent of the cheating problem and the influences on cheating practices as has been discovered in past research, and then examines some of the strategies that have been put forward to address the four conceptual areas. The following is not an exhaustive description of the literature, but does provide a representative sampling of the work in each area.

2.1 The extent of the cheating problem

The literature on cheating in universities report alarmingly high rates of cheating practice and a problem that is long-standing and widespread. For example, an early study of 5,422 North American undergraduate students in 1963 by Bowers (1964) found that 75% admitted to having committed at least one of 13 specific cheating acts. These ranged from copying a few sentences of material without footnoting in a paper (43%) to taking an exam for another student (1%). Another major study in 1993 of 6,096 undergraduate students by McCabe and Trevino (1993) found that 67% admitted to cheating at least once in their course. Around that time a UK study by Newstead, Franklyn-Stokes and Armstead (1996) found that 88% of 943 students admitted to cheating in at least 1 of 21 cheating behaviours, ranging from paraphrasing material without acknowledgement (54%) to sitting an exam for someone else (1%). More recently, Australian studies have reported similar high rates of cheating. For example, in 2001 Marsden, Carroll and Neill (2005) report a study which found that 81% of 954 students admitted to plagiarism and 41% to exam cheating on at least one occasion. Curtis and Popal (2011) report levels of plagiarism of 81% in 2004 and 74% in 2009.

2.2 Factors influencing cheating practice

Cheating practice varies across disciplines, with IT students along with engineering, science and business students engaging in the highest rates of cheating (Bowers, 1964; Davis & Ludvigson, 1995; Roberts, Anderson, & Yanish, 1997). Focusing specifically on IT students, Sheard, Carbone and Dick (2003) found that 79% of 504 undergraduate students admitted to at least 1 of 16 different cheating practices. Other studies have also reported high rates of cheating in IT courses (Barrett & Malcolm, 2006; Simon, 2005).

Studies have identified a number of factors which influence cheating behaviour. These may be personal characteristics, attitudinal or situational factors. The study of IT students by Sheard, Markham and Dick (2003) found time pressure and fear of failing as the main influences. Similar reasons have been found in other

studies. For example, the studies by Newstead et al (1996) and Wilkinson (2009).

2.3 Strategies

As can be imagined, there are many suggestions for strategies to address the cheating problem, often drawn from considering the factors that are believed to influence cheating behaviour. Bennett (2005) proposes that strategies to address cheating should be tailored according to the type of cheating and this research follows that proposal. We now provide an overview of strategies under each conceptual area identified in the Introduction.

2.3.1 Education

Recently, strategies have focused on education about the problems associated with cheating and awareness of policies and possible consequences. A couple of examples of strategies which have been developed for IT students are: an electronic plagiarism tool to educate students about correct use of source material (Barrett & Malcolm, 2006) and a resource to assess IT students' understanding of plagiarism and help them understand how it can be avoided (Joy, Cosma, Sinclair, & Yau, 2009). McCabe (2005) argues strongly that the emphasis should be on using education to develop a culture of academic integrity. Honour codes have been used for this purpose with reported success; however, these are not possible or appropriate for all contexts. Hutton (2006) makes several recommendations about using education to develop a culture of academic integrity to reduce cheating.

2.3.2 Prevention

An important strategy in addressing the problem of cheating is to actively seek ways to prevent cheating. Carroll (2004) comments:

'Catch and punish' approaches are self-defeating in that they absorb huge amounts of staff time, do not lessen the overall incidence of plagiarism, and deflect students from a focus on learning to one devoted to not breaking rules or not getting caught.

Carroll proposes that instead educators should focus on deterrence. McCabe, Trevino and Butterfield (2001) also found that reducing opportunities for cheating was an important tool in reducing academic dishonesty. McDowell and Brown (2001) list a variety of assessment designs that can be used to reduce the opportunity for cheating by students. Davis (1993) provides a useful resource with many ideas for preventing academic dishonesty. Dick, Sheard and Hasen (2008), based on a series of focus groups with 72 IT students, also present a series of assessment design suggestions to prevent and deter cheating by students.

2.3.3 Detection

This area has seen considerable interest over the last ten years with many commercial services being developed such as Turnitin ("Turnitin.com," 2011) and

Blackboard's SafeAssign ("SafeAssign," 2011) along with many individual systems produced by academics and universities to detect cheating, primarily plagiarism. Kohler and Weber-Wulff (2010) have tested the effectiveness of these types of systems to detect plagiarism on several occasions. In the most recent effort in 2010, Kohler and Weber-Wulff tested 47 different plagiarism detection systems. The test found that only 5 of the 47 systems could be classified even as '*partially useful*', with 9 out of 47 being classified as '*barely useful*' and the rest as '*useless*'. Nevertheless, their use has become common in many universities with Turnitin claiming to be used in over 10,000 educational institutions around the world.

In terms of software plagiarism, an area of particular concern to IT academics, a wide range of tools have been developed. Two of the most commonly used are JPlag ("JPlag," 2011) and MOSS ("MOSS: A System for Detecting Software Similarity," 2011).

2.3.4 Consequence

Consequence involves two aspects: what types of consequence are useful in deterring cheating and how to develop a system that manages cases of cheating effectively. Bennet (2005), based on the results of an empirical study, claims that punishment is a deterrent to major plagiarism; however, it is not necessarily effective against minor forms of plagiarism. As well, Genereux and McLeod (1995) found that fear of punishment was one of the most important factors in decreasing planned and spontaneous cheating in their survey of 365 US college students.

Carroll (2002, 2004) and Carroll and Appleton (2001) from Oxford Brookes University provides much guidance on the approaches that universities can take to handle cheating and plagiarism processes. She advocates that effective procedures for managing these issues are ones that:

- *Staff are willing to use and trust*
- *Students experience as fair, transparent, consistent and appropriate*
- *Can be followed without difficulty*
- *Deliver decisions quickly to (potentially) large numbers of students*
- *Produce decisions that can be recorded and defended*

3 Research approach

Students from selected courses in a Faculty of Information Technology were surveyed near the end of second semester 2010. Courses were chosen at each year level of the undergraduate and postgraduate degrees. For the study reported in this paper only data from the undergraduate students were used.

A paper questionnaire was administered in tutorial classes by one of the authors who was not involved in teaching these classes. Participation was voluntary and to

encourage honest responses the questionnaire was anonymous. Most students chose to participate and a total of 117 students from the undergraduate cohort returned completed questionnaires.

Ethics approval for the study was gained from the Monash University Human Research Ethics Committee (MUHREC).

3.1 Survey questionnaire

The questionnaire was developed by the authors and first used for a study in 2000. It was used in the current study with a couple of minor modifications. The questionnaire contained questions to determine:

- demographic information
- students' rating of the acceptability of various questionable work practices described in 18 different scenarios
- students' practice and knowledge of others' practice of each questionable work practice
- factors which could cause cheating
- factors which could prevent cheating

Other questions sought students' responses to the cheating behaviour of other students, and their opinions of staff and University attitudes to cheating. These results have been reported elsewhere (Sheard & Dick, 2011).

3.2 Questionable work practice scenarios

The questionable work practices and factors which could influence cheating were situations which the authors and their colleagues had experienced or were sourced from other studies of cheating, for example, studies by Maramark and Maline (1993) and Newstead et al (1996). To encourage discrimination in ratings of acceptability, the scenarios ranged from practices that would generally not be considered cheating (e.g. showing assignment work to a lecturer for guidance) to serious forms of cheating (e.g. hiring someone to write an assignment). The scenarios were referred to as "questionable" rather than "cheating" practices so as not to prejudice students' judgements of their acceptability. Scenarios to gauge student reactions to cheating have been used in other studies of academic dishonesty, for example, studies by Sierra and Hyman (2008) and Stepp and Simon (2010).

3.3 Analysis

Considering the many possible cheating practices that students may engage in, and the different influences on cheating behaviour, cheating is a complex issue. This study used factor analysis to identify categories of cheating behaviours and influences on cheating behaviour. A similar method was used by the authors in a study of postgraduate cheating behaviour in a previous survey (Sheard & Dick, 2003). Factor analysis is an exploratory technique used to find meaningful structure underlying a number of variables. It is used to reduce a set of variables to a manageable number of dimensions or factors. There are typically two stages to a factor

Scenario	Factor			
	1	2	3	4
Hiring someone to sit an exam for you	.866			
Copying all of an assignment given to you by a friend	.832			
Taking a student's assignment from a lecturer's pigeonhole and copying it	.831			
Using a hidden sheet of paper with important facts during an exam	.774			
Hiring a person to write your assignment for you	.750			
Swapping assignments with a friend, so that each does one assignment, instead of doing both	.732			
Copying another student's assignment from their computer without their knowledge and submitting it	.725			
Using the answer to a tutorial exercise worth 5% by a class mate if the computer you used has problems	.625			
Submitting an assignment based on a friend's assignment from a past running of the subject		.769		
Copying the majority of an assignment from a friend's assignment, but doing a fair bit of work yourself		.655		
Two students collaborating on an assignment meant to be completed individually		.643		
Resubmitting an assignment from a previous subject in a new subject		.452		
Copying material for an essay from a text book	.452		.626	
Copying material for an essay from the Internet	.539		.612	
Obtaining a medical certificate from a doctor to get an extension when you are not sick				.649
Not informing the tutor that an assignment has been given too high a mark				.647
Eigenvalues	5.71	2.73	1.35	1.33
Percentage of variance	31.71	15.19	7.51	7.37

Table 1: Acceptability of cheating: rotated factor matrix

analysis: an *extraction*, which is used to determine the number of factors, and a *rotation*, which is used to obtain a clearer view of the factors thus making these factors more interpretable. The number of factors that are chosen to be interpreted from the extraction depends on whether meaningful interpretations can be placed on the set of factors produced.

Before a factor analysis is performed, it should be determined if the correlation matrix of variables is factorable. This can be determined from the Bartlett's test of sphericity and the Kaiser-Meyer-Olkin Measure of Sampling Adequacy test. If the Bartlett test of sphericity is significant at $p < 0.05$ and the Kaiser-Meyer-Olkin measure of sampling adequacy is greater than 0.6 then we consider that the correlation matrix is factorable.

The factor analysis performed in this study used a Principal Axis Factoring extraction and a Varimax rotation with Kaiser normalization.

4 Results

The analysis of the 2010 survey data to determine influences on undergraduate cheating behaviour was conducted in several stages. First, a factor analysis was conducted on the students' ratings of acceptability of the questionable work practice scenarios to reduce the set of cheating practices to a smaller number of cheating categories. Next, for each category, the students who claimed they had performed any of the practices in the category were determined. This gave a group of cheating and a group of non-cheating students for each cheating category.

Factor analyses were then performed on the reasons for cheating variables and the reasons for not cheating variables, reducing these to smaller sets of *influence* constructs. The mean of each rating was determined within each construct for each student.

Finally, the influence of the reasons for cheating and reasons for not cheating on the cheating behaviour of students within each cheating category was determined by comparing the mean ratings of reasons for each *influence* construct of the cheating and non-cheating groups.

4.1 Categories of cheating

This section explains the process of using a factor analysis to establish a set of cheating categories from the questionable work practices described by the scenarios. For this analysis 16 of the 18 scenarios were used as two scenarios that we do not consider to be cheating practices (i.e. showing assignment work to a lecturer for guidance and posting to an Internet newsgroup for assistance) were not included. For each scenario, the students were asked to rate how acceptable the work practice was using a 5-point Likert scale, where 1 indicates *acceptable* and 5 indicates *not acceptable*. A Bartlett's test of sphericity was significant at $p < 0.05$ and the Kaiser-Meyer-Olkin Measure of Sampling Adequacy test was 0.894, indicating that the dataset was factorable.

An initial factor analysis of the students' ratings of acceptability of the scenarios yielded three factors with eigenvalues¹ greater than 1.0. However, the fourth eigenvalue was very close to 1.0 and an examination of the scree plot² showed a point of inflection between the fourth and fifth factor, indicating that a four factor solution could be investigated. Examination of the variable loadings within the rotated factor matrix of the four factor solution indicated interpretable results for each factor, and this was deemed more interpretable than the three factor solution.

The factor structure for the four factor solution is shown in Table 1. This solution accounted for 62% of the total variance. Using a minimum variable loading of |0.45|, fourteen scenarios show a clear loading on one factor. Two scenarios loaded on factors 1 and 3; however, it was decided to include these scenarios only in factor 3, as their loading was stronger on factor 3 and this made an interpretable factor structure.

The interpretation of the four factors is as follows:

Factor 1: Illegal practices (fraud, stealing)

Factor 2: Collusion (involving assignment work)

Factor 3: Copying (from a book or the Internet)

Factor 4: Deception (administrative – not about the assessment task)

The first three factors could be mapped to the four indexes of cheating found by Lipson and McGavern (1993) in their large study of undergraduate cheating.

4.2 Extent of cheating

For each cheating category established in the previous section, the percentages of students who had performed at least one of the cheating practices was determined. These results are shown in Table 2.

Cheating factor	% students admitting to cheating
Illegal practices	10
Collusion	51
Copying	13
Deception	19

Table 2: Percentages of students in each cheating category

4.3 Acceptability of cheating

For each cheating category, the mean ratings of acceptability were calculated for each student. A

¹ An eigenvalue is a measure of how much variance in the data is explained by a single factor

² A scree plot is produced by plotting the eigenvalues against the factor number

comparison of means between the cheating and non-cheating groups was determined using t-tests. These showed that the students who admitted to have cheated in practices involving collusion or deception found these practices more acceptable than the students who claimed to have not cheating. These results are shown in Table 3.

Cheating factor	Mean acceptability ratings		t-test
	non-cheaters	cheaters	
Illegal practices	4.57	4.57	
Collusion	3.80	2.89	5.55*
Copying	4.42	4.09	
Deception	3.99	3.41	2.90*

* indicates significant difference (p<0.05)

Table 3: Mean ratings of acceptability for non-cheating and cheating groups within each cheating category

4.4 Reasons for cheating

This section explains the process of using a factor analysis to establish a set of influences on cheating constructs from the 14 reasons for cheating. For each reason the students nominated the likelihood that the reason would cause them to cheat using a 5-point Likert scale where 1 indicates *not at all* and 5 indicates *highly likely*. A Bartlett's test of sphericity was significant at $p < 0.05$ and the Kaiser-Meyer-Olkin Measure of Sampling Adequacy test was 0.886, indicating that the dataset was factorable.

An initial factor analysis of the ratings of the likelihood of each reason causing cheating yielded two factors with eigenvalues greater than 1.0. However, the third and fourth eigenvalues were close to 1.0 and an examination of the scree plot showed a point of inflection between the fourth and fifth factor, indicating that a four factors solution should be investigated.

Examination of the variable loadings within the rotated factor matrix of the four factor solution, using a minimum variable loading of |0.45|, indicated interpretable results for each factor. Two variables loaded on both factors 1 and 3; however, it was decided to include these scenarios only in factor 3, as their loading was stronger on factor 3 and this made an interpretable factor structure. One variable "Everybody does it" did not load on any factor. The factor structure is shown in Table 4. This solution accounted for 65% of the total variance. The interpretation of each factor is as follows:

Factor 1: Workload pressure

Factor 2: External pressure

Factor 3: Avoiding failure

Factor 4: Altruism/compensation

The first and third factors which described pressure of workload and concerns about failure were similar influences to those found in studies of undergraduate students by Newstead et al (1996).

Reason	Factor			
	1	2	3	4
Too great a workload at university	.883			
Not enough time	.733			
Will fail otherwise	.669			
Need to get better marks		.737		
Parental pressure		.670		
Lazy		.665		
For monetary or other reward		.536		
Assignments are too hard			.638	
Exams are too hard			.575	
Can't afford to fail	.507		.563	
Afraid of failing	.509		.551	
Missed classes due to ill health				.861
To help a friend				.521
Everyone does it				
Eigenvalues	3.13	2.66	1.84	1.47
Percentage of variance	22.33	18.97	13.12	10.49

Table 4: Reasons for cheating: rotated factor matrix

For each influence on cheating factor, the mean ratings of likelihood of causing cheating were calculated for each student. Comparisons of means between the cheating and non-cheating groups were determined using t-tests. These showed that the students who admitted to having cheated found all types of reasons more likely to cause cheating than the non-cheating students. These results are shown in Table 5.

Reason for cheating factor	Mean likelihood ratings		t-test
	non-cheaters	cheaters	
Workload pressure	2.16	3.27	-5.04*
External pressure	1.46	1.93	-2.72*
Avoiding failure	2.05	2.94	-3.66*
Altruism/compensation	1.79	2.65	-4.10*

* indicates significant difference ($p < 0.05$)

Table 5: Comparison of means of ratings of influences on cheating between the non-cheating and cheating groups

4.5 Reasons for not cheating

This section explains the process of using a factor analysis to establish a set of influences on preventing cheating constructs from the 10 reasons for not cheating. For each reason the students nominated the likelihood that the reason would cause them to not cheat using a 5-point Likert scale where 1 indicates *not at all* and 5 indicates *highly likely*. A Bartlett's test of sphericity was significant at $p < 0.05$ and the Kaiser-Meyer-Olkin Measure of Sampling Adequacy test was 0.797, indicating that the dataset was factorable.

A factor analysis of the ratings of the likelihood of each reason preventing cheating yielded three factors with eigenvalues greater than 1.0. Examination of the

variable loadings within the rotated factor matrix, using a minimum variable loading of $|0.45|$, indicated interpretable results for each factor. This factor structure is shown in Table 6. This solution accounted for 57% of the total variance. The interpretation of each factor is as follows:

Factor 1: Valuing learning

Factor 2: Personal integrity

Factor 3: Fear of consequences

The first factor which describes pride and ownership of work have been found to be the main factors in preventing cheating in studies of undergraduate students (Newstead et al., 1996).

Reason	Factor		
	1	2	3
Pride in your work	.940		
Want to know what your work is worth	.677		
Can get good marks without cheating	.563		
Never thought about it		.699	
Don't know how to		.620	
Fairness to other students		.605	
Against your religious beliefs		.577	
Against your moral values		.496	
Penalties if caught are too high			.859
Fear of being found out			.649
Eigenvalues	2.74	2.07	1.43
Percentage of variance	21.74	20.71	14.31

Table 6: Reasons for not cheating: rotated factor matrix

For each influence on preventing cheating factor, the mean ratings of likelihood of preventing cheating were calculated for each student. Comparisons of means between the cheating and non-cheating groups were determined using t-tests. These showed that personal integrity was a stronger factor in preventing cheating for the students who claimed to have not cheated than the students who had claimed to have cheated. These results are shown in Table 7.

Reason for not cheating factor	Mean likelihood rating		t-test
	non-cheaters	cheaters	
Valuing learning	4.28	3.97	
Personal integrity	3.56	2.84	3.59*
Fear of consequences	3.98	3.78	

* indicates significant difference ($p < 0.05$)

Table 7: Comparison of means of ratings of influences which may prevent cheating between the non-cheating and cheating groups

4.6 Influences on cheating behaviour

The influences on different types of cheating behaviours were explored using the reasons for cheating factors and the reasons for not cheating factors identified in the previous section.

For each of the four categories of cheating behaviour the mean ratings for the reasons for cheating and reasons for non-cheating were compared. These differences were tested using t-tests for independent groups. The results are presented under the four cheating categories:

Factor 1: Illegal practices (fraud, deception, stealing)

This type of cheating involves cheating in exam situations, stealing work from other students and fraud. These practices were seen as the most serious forms of cheating by the students and were the scenarios that had been practised the least. Despite this, 10% of the students in the study admitted to having practiced one of these scenarios at least once.

The students that had performed these practices indicated that there were a number of influences on their cheating. They indicated that they were significantly more likely than the non cheating students to be influenced by external pressures ($t(110) = -2.86, p < 0.05$), and the need to avoid failure ($t(110) = -3.45, p < 0.05$). However, examination of the factors that could prevent cheating showed that there were no differences between the two groups for these influences. These findings are in line with Bennett's study (Bennett, 2005) which found that fear of failure appeared to drive major plagiarism.

Factor 2: Collusion, unacceptable assistance

This type of cheating involves cheating on assignment work. For example, collusion between students on assignment work, submitting a friend's assignment or resubmission of work from a previous running of a subject. This was not seen as a serious form of cheating and 51% of the students admitted to this practice.

The students that had performed these practices indicated that there were many reasons that would cause them to cheat. They indicated that they were significantly more likely than the non cheating students to be influenced by workload pressures ($t(110) = -3.85, p < 0.05$), external pressures ($t(110) = -2.13, p < 0.05$), the need to avoid failure ($t(110) = -2.91, p < 0.05$) and altruism/compensation ($t(109) = -3.84, p < 0.05$). Bennett (Bennett, 2005) also found that minor forms of plagiarism were associated with a wide range of influences.

However, the non cheating students stated that they were significantly more likely than the cheating students to find that personal integrity would influence them not to cheat ($t(111) = 2.56, p < 0.05$).

Factor 3: Plagiarism (copying from a book or Website)

The cheating practices in this factor describe plagiarism where material is taken from books or the Web and not directly from other students. In contrast to the more serious forms of plagiarism described in Factor

2, in these practices the plagiarised material only forms part of the assessment work. The students rated this type of cheating as more serious than the collusion practices in Factor 2 and 13% of the students reported that they had performed one of these practices.

The students that had performed these practices indicated that they were significantly more likely than the non cheating students to be influenced by workload pressures ($t(110) = -2.46, p < 0.05$), external pressures ($t(110) = -3.73, p < 0.05$) and the need to avoid failure ($t(110) = -2.85, p < 0.05$). However, the non cheating students stated that they were significantly more likely than the cheating students to find that personal integrity would influence them not to cheat ($t(111) = 2.23, p < 0.05$).

Factor 4: Deception.

The students that had performed these practices indicated that they were significantly more likely than the non cheating students to be influenced by workload pressures ($t(110) = -2.89, p < 0.05$) and altruism/compensation ($t(109) = -2.53, p < 0.05$). However, the non cheating students stated that they were significantly more likely than the cheating students to find that personal integrity would influence them not to cheat ($t(111) = 2.12, p < 0.05$).

5 Implications for educational practice

The above analysis, especially that of 4.6, gives us leverage in determining where the educator should focus their efforts in terms of our model of the cheating process and in regards to the differing types of cheating. It should be noted that none of these are a 'silver bullet' but they are a means to effectively address the relevant problem.

5.1 Illegal practices

Looking at the first factor of *illegal cheating*, we find that none of the reasons for not cheating have a significant effect on whether a student performs the practice and we see that the two reasons for cheating are the *need to avoid failure* and *external pressures*. In terms of *external pressures*, there is little that can be done in the context of a single subject to affect these. For *need to avoid failure*, some elements of this are amenable to change. Overall, it would seem that a focus on prevention would achieve the best outcomes in reducing this type of cheating, as it reduces the opportunity and/or the benefit of cheating, thereby impacting on those students influenced by *external pressures*. For those students influenced by *need to avoid failure* to cheat in this way, prevention will work by providing relevant scaffolding in the subject so that students do have the resources (time and capability) to succeed without the need for cheating. As well, some emphasis on *detection* is also likely to be necessary, as some students, regardless of *prevention*, will probably attempt this sort of cheating. This is emphasised by the fact that students that practice this type of cheating find it equally unacceptable as non-cheating students, but nevertheless do it anyway.

5.2 Collusion

For the *collusion* cheating type, the factors identified as impacting on this practice are different to those in the first factor. In this case, the reason for not cheating factor *personal integrity* has an impact on the performance of these factors. This indicates that the use of *education* to emphasise the student's need to maintain their *personal integrity* will probably be useful in reducing this type of cheating. The fact that students who perform this type of cheating find it more acceptable to perform than non-cheating students also indicates that there is opportunity to educate them on its unacceptability.

All the reasons for cheating factors impact upon the likelihood of a student cheating in this way. This indicates that this type of cheating will probably be the most difficult to reduce. *Education* will probably be useful in reducing cheating based on the *altruism/compensation* factor, by raising the issue that helping others to cheat is also considered to be cheating. Again, *prevention* is also likely to be useful, as is putting in place a scheme by which *detection* of collusion can be achieved.

5.3 Copying

This factor is very similar to the previous factor, but with the exception that it is not affected by the *altruism/compensation* factor. As such the emphasis on education relating to *altruism/compensation* will probably not be of use in reducing this factor. This is reinforced by the result that cheating students are not significantly different in their level of acceptability for these types of cheating practices than non-cheating behaviour. Though as with the previous factor, education reinforcing personal integrity may help. Similarly as with the collusion factor, a focus on prevention and detection will probably be of value in reducing this type of cheating.

5.4 Deception

The final type of cheating is influenced again by the *personal integrity* factor and the *workload pressures* factor and the *need to avoid failure* factor. This implies that a focus on education to emphasise the personal integrity factor would be of value in reducing this type of cheating. It also implies that prevention by designing into the curriculum reasonable workloads and scaffolding to help students avoid failure would also be useful. As well, *prevention* on the part of the teacher to avoid situations like incorrect marks would also be useful.

5.5 Other implications

An interesting aspect of the results is that the aspect of *consequences* does not seem to provide much if any leverage for the educator to influence cheating practice. For no cheating type does the reason for not cheating *fear of consequences* have any impact on the level of cheating, indicating that a focus on *consequences* such as punishments will not be of value in reducing cheating. Another interesting result was that the reason for not cheating factor *valuing learning* also did not

significantly differentiate between cheaters and non-cheaters for any cheating type, this indicates that cheating practices are probably driven by short-term issues as opposed to a deliberate strategy on the part of students. As such, education based upon valuing learning is unlikely to be effective, in the main, in reducing cheating practices.

In comparison, a similar analysis of a survey of postgraduate students conducted in 2000 showed that *valuing learning* and *fear of consequences* influenced the students not to engage in major and minor forms of plagiarism; however, as for the undergraduate students in this study, there were influences found for preventing serious forms of cheating.

6 Conclusion

In this paper, we have used factor analysis to determine the factors arising from a survey of IT undergraduate students in a number of key areas: cheating practices, reasons for cheating and reasons for not cheating. By determining the relationships between these three sets of factors, it has been possible to determine the specific focus an educator may take to reduce the various types of cheating practices in their class.

7 References

- Barrett, R. & Malcolm, J. (2006) Embedding plagiarism education in the assessment process. *International Journal for Educational Integrity* 2(1).
- Bennett, R. (2005) Factors associated with student plagiarism in a post-1992 university. *Assessment & Evaluation in Higher Education* 30(2), 137-162.
- Bowers, W. J. (1964) *Student Dishonesty and its Control in College* (No. CRP-1672). New York: Columbia University.
- Carroll, J. (2002) *A Handbook for Deterring Plagiarism in Higher Education*. Oxford: OCSLD, Oxford Brookes University.
- Carroll, J. (2004) *Institutional issues in deterring, detecting and dealing with student plagiarism*: JISC briefing paper.
- Carroll, J., & Appleton, J. (2001) *Plagiarism: A Good Practice Guide*: Oxford Brookes University.
- Curtis, G. & Popal, R. (2011) An examination of factors related to plagiarism and a five-year follow-up of plagiarism at an Australian university. *International Journal for Educational Integrity* 7(1), 30-42.
- Davis, B. (1993) *Tools for Teaching*. San Francisco: Jossey-Bass.
- Davis, S. F. & Ludvigson, H. W. (1995) Faculty Forum: Additional data on academic dishonesty and a proposal for remediation. *Teaching of Psychology* 22(2), 119-121.
- Dick, M., Sheard, J., Bareiss, C., Carter, J., Joyce, D., Harding, T. & Laxer, C. (2003) Addressing student cheating: Definitions and solutions. *ACM SIGCSE Bulletin* 35(2), 172-184.
- Dick, M., Sheard, J. & Hasen, M. (2008) Prevention is better than cure: Addressing cheating and plagiarism

- based on the IT student perspective. In T. S. Roberts (Ed.), *Student Plagiarism in an Online World: Problems and Solutions* (pp. 160-182). Hershey, PA, USA: Information Science Reference.
- Genereux, R. L. & McLeod, B. A. (1995) Circumstances surrounding cheating: A questionnaire study of college students. *Research in Higher Education* 36(6), 687-704.
- Hutton, P. (2006) Understanding student cheating and what educators can do about it. *College Teaching* 54(1), 171-176.
- Joy, M., Cosma, G., Sinclair, J., & Yau, J. Y.-K. (2009) *A taxonomy of plagiarism in computer science*. In proceedings of the EDULEARN09, Barcelona, Spain.
- JPlag. 2011. <https://www.ipd.uni-karlsruhe.de/jplag/> [accessed 26 August 2011]
- Köhler, K. & Weber-Wulff, D. (2010) Plagiarism and Detection Test.
- Lipson, A. & McGavern, N. (1993) *Undergraduate academic dishonesty at MIT. Results of a study of attitudes and behaviour of undergraduates*. In proceedings of the Annual Forum of the Association of Institutional Research, Chicago, USA.
- Maramark, S. & Maline, M. B. (1993) *Academic dishonesty among college students. Issues in education*. (Information analyses No. OR-93-3082). Washington, DC: Office of Educational Research and Improvement (ED).
- Marsden, H., Carroll, M. & Neill, J. (2005) Who cheats at university? A self-report study of dishonest academic behaviours in a sample of Australian university students. *Australian Journal of Psychology* 57(1), 1-10.
- McCabe, D. L. (2005) Cheating among college and university students: A North American perspective. *International Journal for Educational Integrity* 1(1).
- McCabe, D. L. & Trevino, L. K. (1993) Academic dishonesty: Honor codes and other contextual influences. *Journal of Higher Education* 64(5), 522-538.
- McCabe, D. L., Trevino, L. K. & Butterfield, K. D. (2001) Cheating in academic institutions: A decade of research. *Ethics & Behavior* 11(3), 219-232.
- McDowell, L. & Brown, S. (2001) *Assessing students: cheating and plagiarism: The Higher Education Academy*.
- MOSS: A System for Detecting Software Similarity. 2011. <http://theory.stanford.edu/~aiken/moss/> [accessed 26 August 2011]
- Newstead, S. E., Franklyn-Stokes, A. & Armstead, P. (1996) Individual differences in student cheating. *Journal of Educational Psychology* 88(2), 229-241.
- Roberts, P., Anderson, J. & Yanish, P. (1997) *Academic misconduct: Where do we start?* In proceedings of the Northern Rocky Research Association, Jackson, Wyoming.
- SafeAssign. 2011. <http://www.safeassign.com/> [accessed 26 August 2011]
- Sheard, J., Carbone, A. & Dick, M. (2003) *Determination of factors which impact on IT students' propensity to cheat*. In proceedings of the fifth Australasian Computing Education conference, Adelaide, Australia.
- Sheard, J. & Dick, M. (2003) *Influences on cheating practice of IT students: What are the factors?* In proceedings of the eighth Annual conference on Innovation and Technology in Computer Science Education, Thessaloniki, Greece.
- Sheard, J. & Dick, M. (2011) *Computing student practices of cheating and plagiarism: A decade of change*. In proceedings of the 16th Annual conference on Innovation and Technology in Computer Science Education, Darmstadt, Germany.
- Sheard, J. Markham, S., & Dick, M. (2003) Investigating differences in cheating behaviours of IT undergraduate and graduate students: The maturity and motivation factors. *Journal of Higher Education Research and Development* 22(1), 91-108.
- Sierra, J. & Hyman, M. (2008) Ethical antecedents of cheating intentions: Evidence. *Journal of Academic Ethics* 51-55.
- Simon. (2005) Electronic watermarks to help authenticate soft-copy exams. *Australian Computer Science Communications* 27(5), 7-13.
- Stepp, M. & Simon, B. (2010) *Introductory computing students' conceptions of illegal student-student collaboration*. In proceedings of the SIGCSE'10, Milwaukee, Wisconsin, USA.
- Turnitin.com. 2011. <https://www.turnitin.com/static/index.php> [accessed 26 August 2011]
- Wilkinson, J. (2009) Staff and student perceptions of plagiarism and cheating. *International Journal for Educational Integrity* 20(2), 98-105.

Why the bottom 10% just can't do it - Mental Effort Measures and Implication for Introductory Programming Courses

Raina Mason¹

Graham Cooper¹

¹ Southern Cross Business School
Southern Cross University
Hogbin Drive, Coffs Harbour, New South Wales 2450

raina.mason@scu.edu.au
graham.cooper@scu.edu.au

Abstract

This paper reports the results of mental effort measures and comments collected as part of a study of 44 introductory programming courses in 28 Australian universities, conducted in the latter months of 2010. Academic staff were interviewed regarding their perceptions of the mental effort that is required by themselves, an average student, and a low-performance student while attempting to solve and learn from a novice programming problem.

Qualitative responses were also gathered from academics to gain insight into the various student profiles and impediments to learning for low-performing students. Mental effort results indicated that many low-performance students typically experience high to extreme levels of mental effort. Verbal responses obtained from academics also indicate an awareness that for many low-performance students learning fails due to excessive demands being placed upon their cognitive resources.

It is suggested that for many low-performance students learning fails due to cognitive overload. The implications for the selection of languages and environments and for the design of introductory programming courses (units) are discussed.

Keywords: introductory programming, programming success, pedagogy, Australian university courses, mental effort measures, cognitive load, cognition.

1 Introduction

A survey of introductory programming courses (units) offered in Australian universities was conducted at the end of 2010. This was designed to repeat similar studies regarding introductory programming courses within Australian universities that had been conducted in 2001 (de Raadt, Watson & Toleman, 2002) and 2003 (de Raadt, Watson & Toleman, 2004). The findings of the 2010 survey are reported in Mason et al. (In press).

The primary focus of the 2010 survey was to identify the programming language(s) used within Australian university courses, along with teaching-learning support factors such as development environment, textbook used

(if any), hours of teaching and focus upon problem solving. Differences between the 2001/2003 censuses and the 2010 study are suggestive of trends in approaches to teaching introductory programming within Australian universities. The major changes were shifts towards some languages (Python and C#) at the expense of other languages (Visual Basic and C++). There was also an increased usage of IDEs and other tools. Discussion was also provided regarding perceived reasons offered by interviewees (academics teaching introductory programming courses) for such changes, which were predominately reported to be pedagogical.

A secondary focus of the survey was to investigate the evaluations by introductory programming academics of the level of “mental effort” that would be required by students studying in these programming courses. The primary outcome of the study in this regard is that academic staff instructing in introductory programming courses rated their own levels of mental effort in dealing with the course (unit) content to be “low”, the mental effort of average students to be “above average”, and the mental effort of students in “the bottom 10% of performance levels” to be “very high to extreme”.

The current paper further explores the role of mental effort within the cohort of “the bottom 10% of performance”.

2 The Role of Mental Effort in Learning

2.1 Mental Effort

Mental effort is a cognitive construct that has its derivation in Cognitive Load Theory (Sweller 1999) and refers to the deliberate focus of attention and level of cognitive resources that are allocated to a task (Kirschner 2002).

Mental effort is closely related to cognitive load. Cognitive load is a “count” of the number of elements that must be held within working memory for any particular processing task (Sweller 1999). Each element will be in the form of a schema (Chi, Glaser & Rees 1982) as it exists for an individual based upon their current personalised knowledge base. Schemas are well organised hierarchical networks of conceptual and procedural information. As ones’ expertise in a content domain expands, organises and intercepts, so too does ones’ schemas become larger, more well organised and more interrelated. A novice in a content domain only holds small disconnected schemas, and so needs to hold a larger number of elements in working memory, than does

Copyright © 2012, Australian Computer Society, Inc. This paper appeared at the Fourteenth Australasian Computing Education Conference (ACE2012), Melbourne, Australia, January 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 123. Michael de Raadt and Angela Carbone, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

an “expert” in the area, when consciously attending to the same body of content.

“Mental effort” is a term considered to be more accessible to lay persons due to its alignment to a general enquiry regarding “how hard one is thinking”, rather than a more load-orientated enquiry of “how many things one is thinking about”. Mental effort has historically been used as a measure to evaluate cognitive load (Paas & Van Merriënboer 1993).

2.2 The Targets of Mental Effort

Not all content is equally challenging to learn. There are three distinct factors which may contribute to a student’s experienced ease or difficulty in learning a body of to-be-learned content. These may be identified as intrinsic, extraneous and germane factors.

Some content, by its *intrinsic* nature, is considered to be “difficult”, and thus requires a high level of mental effort to comprehend and learn. This is primarily due to the inherent complexities, interrelations and subtleties of the information. Both mathematics and computer programming are such intrinsically difficult areas (Sweller & Chandler 1994).

In the context of learning computer programming there are at least five distinct domains that must be mastered at the same time. These have been identified by du Boulay (1989) as: general orientation, the notional machine, notation (syntax and semantics), structures, and pragmatics (developing, debugging, etc). The current paper argues that this multi-domain aspect of programming is a primary contributor to its reputation as being difficult to learn.

Another potential source of difficulty in learning content lies in the way in which to-be-learned information is presented. This is *extraneous* to the conceptual understanding of content, yet needs to be processed by a learner’s cognitive resources because it is the means by which information is presented to the learner. On this basis it may be the “materials”, or “media”, or “activities” involved in the teaching-learning transaction (Ayres & Sweller 2005).

For example, text based instructions specifying how to travel from one location to another are usually more difficult to understand than an equivalent-content source based around the use of visual maps. This is true regardless of whether one is travelling from home to a friend’s house, or from one city to another. Maps have an obvious spatial relation to the 2-dimensional landscape which they represent, while the textual form does not.

Similarly, different programming languages and environments may present differing levels of visual representations of core programming concepts, such as loops (iterations). The choice of computer programming language and/or environment may have a direct impact upon the ease of comprehending, learning and applying underlying programming concepts. The language itself is extraneous to understanding these core programming concepts, yet it is the primary vehicle by which these are usually presented to a student. A student who does not hold schemas for core programming concepts and who falters on aspects of language or syntax, will likely be unable to distinguish between such core concepts versus aspects of the language, as to why he or she has faltered.

The third factor that gives rise to mental effort is from the conscious focus of attention that a learner brings to a task to deliberately “remember” and/or “understand” to-be learnt content. In addition to this, a learner may also seek to proceduralise this newly acquired knowledge base through strategies such as drill-and-practice and mental rehearsal (Cooper, Tindall-Ford, Chandler & Sweller 2001). In this context mental effort is germane to the tasks of schema acquisition and automation (Paas & Van Merriënboer 1994). These activities generally consist of making mental comparisons and contrasts between one’s already-held knowledge base (their schemas) and the newly presented information, along with some form of practice to proceduralise the application of these evolving schemas. A broad instructional strategy that benefits learning is to maximise the level of cognitive resources dedicated to such germane activities (Paas, Renki & Sweller 2003).

Depending upon the nature of the to-be-learned content germane activities may range from shallow processing tasks such as rote-learning to deep processing elaboration strategies. For example, consider a student studying chemistry. The elements of the periodic table may be rote-learned as a simple sequence, or more deeply processed and linked to other facts that may already be held, or developing, such as atomic weight, valency and chemical reactions.

Similarly, programming may be viewed as a set of isolated base concepts to be rote-learned, or analysed, abstracted and re-organised into a wide range of algorithms that are utilised within programming structures. In general, the deeper the level of processing, the more able will the learner be in recalling and applying the information due to the level of “understanding” that has taken place, rather than only memorisation.

The bottom line, from a Cognitive Load Theory perspective (Paas, Renki & Sweller 2004), is that these three demands of mental effort compete for cognitive resources. An issue arises because human cognitive resources are strictly limited (Miller 1956) and so the combined demands of these three targets of mental effort cannot always be met. The reality is that students, in attempting to deal with the broad complexity of programming, in conjunction with a specific programming language and/or environment, may have insufficient cognitive resources left available to dedicate to the primary purpose of the teaching and learning transaction - that of actually engaging in germane tasks such as comprehending, learning and proceduralising the newly presented to-be-learned content regarding core programming concepts.

It is important to note that these potential difficulties described in comprehending and learning may be greatly amplified for students at the lower end of the performance spectrum.

The next section briefly outlines some of the key features of human cognitive architecture and Cognitive Load Theory, before returning to the findings of the 2010 survey, and the inclusion of questions regarding mental effort.

3 Human Cognitive Architecture

3.1 Cognition, Memory, Schemas and Automation

Human Cognitive Architecture is often described as an information processing model with operational relations between the separate conceptual functions of Long Term Memory and Working Memory (Sweller 1999). There are added complexities to the model with the inclusion of Sensory Memory to handle, very briefly, the input from our senses, and modality specific slave-systems for management and integration of distinct visual and auditory information components within Working Memory (Mayer 2005). While these have significant implications for the processing of instructional materials, they will be left largely unexplored in the current paper.

Long term memory is a virtually permanent store of knowledge. It is effectively unlimited in its capacity, with knowledge stored in the form of “schemas” which are hierarchically structured information networks that gather and interrelate throughout a person’s lifetime (Sweller 1999).

In contrast, Working Memory is dedicated to the conscious processing of information. It is where activities such as thinking and problem solving take place. The elements of knowledge, which are held in the Long Term Memory store as schemas, are utilised within working memory processing tasks (Sweller 1999).

Working memory is strictly bounded in its capacity with 7 elements (+/-2) representing typical performance by adults on random memory tasks (Miller 1956). This number will be further reduced when attending to elements that are not random, but hold well-defined relations. For example, the formula “ $F=ma$ ” does not only require holding of the concepts of force, mass and acceleration, but also their relation as defined by the formula.

The limitations of Working Memory may be worked-around by humans through selecting “larger” size elements to process. These elements are the schemas that one already holds in Long Term Memory. For example, the schema for “alphabet” can easily be decoded into a set sequence of 26 letters that most readers will well know.

A second primary performance aspect of human cognitive processing models is “automation”, which refers to a person’s ability to proceduralise and apply their schemas while needing only very low levels of conscious attention (Shiffrin & Schneider 1977).

Models of expertise typically reduce the primary traits of expertise to the possession of relatively large well-structured networks of schemas, and the capacity to access and apply the knowledge and skills held in these schemas, with low levels of conscious attention. These two attributes are sufficient to explain virtually all aspects of expert performance, including transfer and problem solving (Cooper & Sweller 1987).

Cognitive Load Theory (Paas, Renki & Sweller 2004) explores the application of information processing models to the processes of learning, and more importantly, to the design of instructional materials and activities, with the purpose of facilitating learning. Central to this approach of instructional design is the limitations of working

memory. If the mental processing load placed upon these limited resources are exceeded at any point during a learning transaction (“overloaded”), then learning will falter due to the dropping of information that was currently being processed.

There are many apparent similarities between this model of human cognitive architecture and its manner of information processing, with those associated with computer architectures and data processing. This is not accidental. There have long been identified relations between these two areas that have effectively evolved in parallel (Hunt 1982). If you (the reader) have a high knowledge base in computer programming and architecture, but a relatively low knowledge base in human cognitive architecture, then you may think of “cognitive overload” as being akin to a “buffer overflow”. Overloading means that some data is dropped. Once dropped, it is lost -irretrievably - and so while some form of processing may still occur, the nature of the information now being handled is incomplete, or worse still, incorrect and can result in an effective program crash, or memory access errors.

3.2 Sources of Cognitive Load

There are three identified sources of cognitive load. These are Intrinsic, Extraneous and Germane (Paas, Renki & Sweller 2003). Each was described in lay form earlier in this paper in the context of mental effort. Each is dealt with again from a more technical perspective.

3.2.1 Intrinsic Cognitive Load

Intrinsic cognitive load refers to the innate difficulty of a body of to-be-learned content. Information that contains many internal relationships between knowledge elements (element interactivity) imposes a higher level of cognitive load than information that can be considered to be a series of simple, unrelated facts (Sweller & Chandler 1994). This is because it is not only the separate discrete elements that need to be consciously attended to, but also the relations between those elements. A popular example to consider is the learning of a second language -- not a second programming language, but a second spoken language. Assume that you, the reader, are relatively expert at English, and learn Indonesian, French or Spanish. Each is written in a similar script to English and has a large level of overlap in phonemes.

In learning a second language (or a third, fourth and so on), there at least two distinct aspects that need to be learned. These are the vocabulary of the language and the syntax (or grammar) of the language. The vocabulary is relatively simple, being in general terms a one-to-one word-swap for “me”, “you”, “good”, “bad”, “morning”, “evening” and so on. There are several hundred, perhaps thousand, such word-swaps required in order to be able to communicate meaningfully at a conversational level. The second aspect, the syntax, is far more difficult to learn, because to construct a sentence such as “How are you this morning?” is generally not just a process of swapping words. All of the words may need to be considered simultaneously, along with their inter-relations. Note, too, that this is a simple, common sentence! If there are more subtle aspects being included such as personal

relationship, respect associated with maturity (age of the persons involved), time of day being said, and the gender of the items (not the people, but the items such as table, chair, food) in question (of which there is no meaningful equivalent in English) then the translation can become much more difficult if there is a requirement to “be grammatically correct”.

A question such as “Have you eaten dinner this evening?” needs much more than a simple word-swap process to be translated into many languages because the order of the words may need to be re-sequenced, there are likely issues of verb conjugation, there is a need to be mindful of the “respect-relationship” between the players (akin to using “sir”, “Mr” or “mate” in the Australian context), and we have not even yet considered aspects of enunciation and emphasis.

The purpose of this discussion of language is to indicate that learning a second language contains a mix of intrinsically relatively “easy” content - the one-to one word swaps - plus the relatively “difficult” content of syntax, which is difficult due to the element interactivity between the separate words. On top of this there may be physical performance difficulties such as recognising and producing specific sounds (phonemes). We will return to the issue of learning a language later in this paper, although the context will then move to that of computer programming languages.

3.2.2 Extraneous Cognitive Load

Extraneous cognitive load refers to the load generated by the format of instructional materials and/or to the performance of learning activities. Some formats and activities hinder learning by loading the learner with unnecessary information processing (unnecessary to the task of acquiring schemas and automating them). Extraneous cognitive load is open to being manipulated by the instructional designer. There are several common effects that have been investigated by Cognitive Load Theory. Two that are relevant for discussion here in the context of introductory programming are the Worked-Example Effect (Cooper & Sweller 1987) and the Split-Attention Effect (Chandler & Sweller 1992).

Many studies have indicated that a relatively high weighting upon problem solving tasks is less effective at facilitating learning than studying a set of similar worked examples (Renki 2005). The underlying theoretical analysis is based upon the argument that novices use means-ends-analysis to solve problems, that this process imposes a relatively high level of cognitive load, directs attention to the answer, rather than the process of obtaining the answer, and consequently impedes learning (Sweller 1988). It was observed though the survey that many introductory programming courses still place a relatively high emphasis upon problem solving. By this we mean that a core, central activity within the learning materials, is to require students to be actively engaged in producing a solution to a computer programming problem solving task. Such a focus on problem solving may hinder learning.

The Split Attention Effect (Chandler & Sweller 1992) argues that split formats of instruction, whereby two or more mutually referring sources of instruction are presented in isolation, and thus need to be mentally

integrated to enable comprehension, will be less effective at facilitating learning than an equivalent-content integrated format where all mutually referring sources of information have been integrated into a single source.

For example, consider a split attention format that may result from a physical layout of instructional materials that “splits” the location of instructional text from the location of a text-referenced diagram which usually sits beside or below the text. This will impose a level of visual search and require a mental integration of meaning between the textual and graphical components of instruction to enable comprehension. This search and mental integration is not required for an equivalent-content presentation where the textual information has been integrated (embedded) within the diagram. The added cognitive load due to the visual search and mental integration of the two mutually referring sources of information in a split format causes additional taxing of cognitive resources, and so reduces the cognitive resources available to be applied to the germane activities associated with schema acquisition and automation. As a direct result of such an instructional design, learning is reduced. This effect has been demonstrated to be present in many instructional settings and is highly robust (Sweller 1999).

We will return later to the split attention effect, not to discuss the physical integration of materials, but to discuss the split-attention between learning underlying concepts of computer programming compared to the learning of a specific computer programming language (along with syntax). A common feature of introductory programming courses appears to be that the core programming concepts are “cocooned” within a specific computer programming language and syntax.

3.2.3 Germane Cognitive Load

Germane cognitive load refers to load devoted to the processing, construction and automation of schemas. This application of cognitive load is beneficial to learning. A general strategy in many instructional settings is to reduce extraneous load, and to direct the released cognitive resources towards the germane efforts associated with schema acquisition and automation (Paas, Renki & Sweller 2004).

In situations where both intrinsic and extraneous loads are high it is likely that this will effectively block the capacity for germane load, and thus block learning.

It is argued in the current paper that this is the situation experienced by many low performing students who are undertaking introductory programming courses. These students, who lack schemas in programming, are faced with the intrinsic complexity of core programming concepts embedded within an instructional presentation that involves aspects of computer language and syntax which are extraneous to this task. It is likely that these students will have insufficient cognitive resources available to enable the process of learning.

4 Cognitive Load and Novice Programming Problems

The current survey explored aspects of cognitive load associated with the teaching and learning of programming

concepts within introductory programming courses offered by Australian universities.

Specifically, in the context of learning the generic concepts of programming through completing computer programming problem solving tasks, there are three separate components of processing that need to be attended to, and it is argued that these broadly align to the three different sources of cognitive load. These are:

- Intrinsic load, associated with the concepts and interpretation of problem statements;
- Extraneous load, determined by the language and environment, along with associated constraints such as syntax; and
- Germane load, associated with the cognitive processing to acquire and automate new schemas.

The term “mental effort” has been used to communicate with participants rather than “cognitive load”. This self-reported assessment of perceived mental experience is taken as a measure of the cognitive load associated with a task performance.

Participants in the survey were asked to rate their own levels of mental effort on each of the three components of cognitive load, using a 9 point Likert scale (where 1 = ‘no mental effort’ and 9 = “extreme mental effort”). Participants were also asked to estimate the levels of mental effort on each of these components experienced by an average student in their introductory programming course and that experienced by a student in the ‘bottom 10% of performance’ in their course.

There were some participants who did not immediately offer a response for all, or an aspect, of this series of questions, particularly for the “bottom 10% of performance”. These were removed from the first series of analyses, but will be commented upon further below.

An initial analysis was performed on the mental effort scores given by participants (instructors), for each of these three components of cognition -- understanding the problem statement, using the environment, and reinforcing previous concepts. A series of comparisons between, firstly, the self-rating of the participant (a first year programming instructor) and the rating anticipated to be experienced by an ‘average student’; and then between the anticipated average student’s rating and the anticipated level to be experienced by a student in the bottom 10% of the class’s performance. The comparisons were performed using a series of Wilcoxon Signed Rank tests.

These results were reported in Mason et al. (In press), and are replicated here to enable further analysis and discussion. Table 1 shows the mean, median and mode for each of these cognitive load areas for instructors, the average student and students “in the bottom 10%”. The data was heavily skewed, so measures of central tendency provided are modes and medians.

Table 2 below shows the results of the Wilcoxon Signed-rank tests (one-tailed) for within-subject comparisons between instructors and average students, and between average students and students in the bottom 10%.

		instructor	average student	bottom 10% student
intrinsic	mode	2	6	9
	median	2	6	8.5
extraneous	mode	2	5	9
	median	2	5	8
germane	mode	2	7	9
	median	2	5.5	8.5

Table 1: Levels of mental effort reported

	Instructor < average student (greater mental effort)				
	W	Ns/r	z	p	n
Understanding and processing the problem statement	811	43	4.39	<0.0001	43
Navigating/using the environment, tools or language	838	41	5.43	<0.0001	43
Learning from the problem/ reinforcing previous concepts	647	40	4.34	<0.0001	42
	Average < bottom 10% student (greater mental effort)				
	W	Ns/r	z	p	n
Understanding and processing the problem statement	207	24	2.95	0.0016	25
Navigating/using the environment, tools or language	276	23	4.19	<0.0001	25
Learning from the problem/ reinforcing previous concepts	144	20	2.68	0.0037	21

Table 2: Wilcoxon Signed-Rank Tests

These results indicate that for each of the three sources of cognitive load, the instructors in the introductory programming courses rated their own levels of required mental effort to be low, and that they expected that average students would need to exert higher levels of mental effort than themselves - ‘above average’.

Additionally, again for each of the three sources of cognitive load, the participants rated the anticipated mental effort to be experienced by a student in “the bottom 10%” to be higher than that of an average student, rating at very high to extreme mental effort.

These results are consistent with the argument provided earlier regarding cognitive overload. Students who are in the bottom 10% of performance are perceived by the introductory programming instructors to be effectively swamped in mental effort on each of the three measures. As these three areas compete for cognitive resources it is unlikely that students can allocate such high levels to all three components, despite their desires to do so, and despite the instructors perceptions that students are attempting to do so.

Human cognitive processes are limited. Student have no option but to try and understand and respond to the programming problem statement, as this is the primary task to which they have been explicitly directed to engage with. The student also has no option but to navigate, as best he or she can, the programming language and syntax,

as this is the environment which they have explicitly been directed to operate within. Given that these two tasks are each placing very heavy processing burdens upon mental resources it may be expected that the third area, those of germane activities, are effectively strangled for processing resources.

Recall that not all participants gave a score for the mental effort measures, instead choosing to provide comment. The second layer of analysis, now pursued, focuses upon the verbal comments provided by all participants regarding their perceptions of mental effort for the different student cohorts.

4.1 Participants that did not give a score for all or any of the cognitive load aspects

A total of 44 participants took part in the survey, and all of these participants answered the questions regarding the mental effort experienced by themselves on each of the three factors while solving a novice programming problem. One of the participants declined to answer the questions regarding average and 'bottom 10%' students as he/she indicated that he/she believed "that the 'average student' could be designated as any of the scores on the Likert scale" We consider this to indicate a misinterpretation of our line of enquiry, and no further consideration is given to the responses provided by this participant. Of the 43 remaining participants, one declined to give a score for the "germane" aspect for the "average student", whilst offering the scores for the other aspects.

Only 25 participants gave any scores for the "bottom 10%" and often this was coupled with qualifying statements. Only 21 of these participants were willing to specify a score for mental effort for the germane aspect for students in this bottom 10% of performance in the course.

5 Participants' Comment Analysis

Many participants indicated that providing an estimate of mental effort for lower performing students was a much more complex scenario than could be expressed by a simple numerical answer to a Likert scale question. Even those who did give a score often added comments to qualify their answer.

5.1 Methodology of further analysis

The phone interviews with the 44 participants in the survey were audio-recorded with their permission, with the exception of one participant, who agreed to be surveyed but did not give permission to be recorded. For this particular interview, comprehensive notes were taken and recorded by hand on the interview script. The remaining audio-recordings were then transcribed.

Truncated versions of each interview transcript were created for the purposes of further analysis, temporarily disregarding data on languages, textbooks, on-campus hours and other aspects of introductory programming courses within Australian universities, which have been reported elsewhere (Mason et al. In press). Data retained included mental effort measures for "students in the bottom 10% of the course" (where given), any transcribed comments that offered insight into mental effort (or lack

of such) used by students, and any other comments that were offered by participants throughout the course of the interview that had bearing on possible reasons for success or failure of students in the bottom 10% of the course.

Interviews were confidential and participants were encouraged to be open, frank and fearless. On this basis the researchers have redacted phrases that may be considered to be disrespectful of students as clients of the education system. Thematic analysis of the comments has been performed, but in some cases specific terminology has been withheld. In these cases, "[redacted]" has been used. This indicates terminology (not profanity) that may be interpreted as offensive to the people thus identified and labelled.

5.2 Student Profiles

This survey asked for Likert scale measures for mental effort for students in the bottom 10% of course performance. If the instructor asked the interviewer for clarification of type of student targeted, then the interviewer (using the terminology used by the instructor) indicated that the mental effort expended by a less capable student who is trying was required.

[Instructor] "by effort do you mean students that apply themselves that are [redacted]? Or students that don't apply themselves?"

[Interviewer] "apply themselves that are [redacted]"

However, some instructors offered two values (one for students who did not try, and those who were trying), and others stated that one simple measure couldn't be given and instead offered comments. For this reason, the comments associated with all of the types of student profiles, as well as mental effort measures for the student profile of "students who are less capable but are trying", have been analysed.

The interview summaries were examined for commonalities between participants regarding the profiles of students composing the bottom 10% of their course. Responses indicated that 34% (15/44) of participants explicitly indicated that more than one student profile existed in the student composition of the bottom 10% of their course. All participant summaries were examined to identify the range of profiles of these students.

The profiles of students that were identified by instructors were:

- 'Strivers' - less capable students who are actually trying - identified by 75% of participants;
- 'Idlers' - those students who attend class but who do not try - identified by 40% of participants;
- 'Ghosts' - students who do not attend class/ are not seen by instructors - identified by 14% of participants.

Each profile type was then examined separately using thematic analysis across participants to determine possible reasons given by participants for lack of success by these students.

5.2.1 Ghosts - students who are not there

A relatively small number of participants (14%) identified that some or all of the students in the bottom 10% of the course were never seen by the instructor, or

disappeared early. We have designated this cohort of students “Ghosts”. Reasons offered for this disappearance revolved around the students’ perceived lack of motivation and desire to participate in the course or their perceived lack of choice in studying programming as part of their degree structure. Participants have been identified throughout this paper by transcript number -- for example [1] -- to show that comments have been sourced from a range of participants. Comments provided regarding the “Ghost” cohort include:

- “Enrolled because [they] “have to” - they are required to be enrolled full-time. Students are embarrassed to tell parents they’ve made the wrong decision about a university course (as opposed to a TAFE course) and just stop going. Only 1% of students choose to do the unit, for the rest its core. They don’t really want to be there.” [25]
- “they don’t want to come onto a university campus, they see this as an enormous burden that gets in the way of their crappy part-time job, and basically their commitment is (on average) pathetic” [31]
- “it’s coming back to the students’ desire and motivation to learn, and that’s the main problem we are having at the moment. I don’t think that teaching or learning programming is a difficult thing, I think it’s the students that are the problem at the moment.” [2]

5.2.2 Idlers - students who do not apply themselves

Instructors identified two separate profiles of students who did attend classes or were ‘seen’, within the bottom 10% of students. A cohort of “students who attend class but don’t try” were identified by 40% of participants. We have designated this profile of students as “Idlers”. “Strivers” - the remaining profile of students - are examined in the next section of this paper.

Some of the reasons offered by instructors for the Idlers’ lack of progress were similar to those offered for the Ghosts.

Students’ lack of motivation and desire to learn was given as a reason in many cases:

- “... there is very little mental effort because they don’t do anything. In workshops they are on Facebook or chatting or getting the answers from their mates rather than working it out for themselves.” [5]
- “Don’t try. Hand in other people’s assignments, don’t attend, don’t do tasks, are on facebook.” [23]
- “not motivated enough to try” [26]

Several instructors offered the reason for this lack of effort to be that some students were forced to do the unit by their degree requirements, had no intent of going further, did not enjoy programming and that these students didn’t apply themselves to learning:

- “Only about 20% of students [in the course] intend to progress with programming” [23]

- “Students enrol and discover they don’t like programming, so the effort they expend is minimal” [28]

Mason et al. (In press) reported on the falling numbers of students in programming courses and ICT programs as a whole. Some universities have responded to this trend by lowering entrance score cut-offs, and instructors identified this as a source of “Idlers” those students who perhaps were not suited to university study, but had managed to enter the course:

- “... quite a low entry bar into our computing degrees” [29]
- “we have ‘OP [Overall Position Tertiary Entrance Rank] ridiculous’ as a large cohort” [31]

More concerning were the comments offered by some instructors that indicated that these “Idlers” were predisposed to believe it was impossible to succeed, or became frustrated so early in the course that they gave up:

- “Students come in and say "I've heard programming is so bad and its the end of the world" ... And so they are *defeated before they get here* [emphasis added]. But they just don't do any work ...They are completely directionless.”[34]
- “there are some students who are too frustrated with it to really try that hard, and dismiss the learning.” [38]

5.2.3 Strivers - students who are less capable but are trying

Most (75%) instructors indicated that the bottom 10% of students in their course contained students who were less capable but were trying to succeed. We have designated these students “Strivers”. Recall from Table 1 that the mode on each of the three mental effort measures for the students in the bottom 10% was reported as “9”, the maximum value available on the Likert scales.

Instructors also often offered indicative comments about the level of mental effort for Strivers, regardless of whether they gave a Likert scale number. These comments are offered below to show the general agreement that for this profile of students the mental effort required for all three aspects was very high to extreme, and in some cases “off the scale”. These include:

- “off the scale” [11]
- “putting all their effort in” [15]
- “try very very hard” [16]
- “expend more energy for borderline pass - more effort in getting that borderline pass than others use to get higher marks” [28]
- “very high level of mental effort” [30]
- “factor of 10 at the very minimum” [34]
- “try but can’t solve the problem at all” [35]
- “a lot more effort” [41]

Given that these Strivers are often being reported as trying as hard as they can, it is important to consider the reasons given for why they find it so difficult, or fail (if any reasons were given).

One group of instructors offered the opinion that students had an innate aptitude (or lack of aptitude) for

programming and that if this was missing, those students would never understand programming - no matter how hard they tried. These comments are presented below:

- “some of them pick it up quickly, some take longer and some never pick it up at all.” [7]
- “they struggle - they genuinely try and they put a lot of work in but it’s not how their brains are wired” [22]
- “There are those that get it ... and some people for some reason just don’t get it and they are hopelessly lost, and they just never seem to be able to get it at all.” [23]
- “some students have some sort of mental block as far as programming is concerned. They might find it difficult to even follow a set of sequenced step-by-step instructions.” [27]
- “they try really hard but the penny doesn’t drop” [22]

Other instructors commented that the Strivers lacked literacy, comprehension and analysis skills prior to entering the introductory programming course, and that this compounded the difficulty of learning the material for this cohort:

- “they have very poor literacy and comprehension skills” [15]
- “[Programming] is really a bit harder than the other units - it requires some analysis and some mathematical nous almost. And they simply don’t have it.” [34]

Strivers from a non-English-speaking background had a particularly hard time, according to some participants. Some of these students may be more capable (than their programming performance indicates) but the additional cognitive load imposed by working in English as a second language means that they fail to learn or perform:

- “high level of overseas students - sometimes what seems plain to us is not plain to them” [27]
- “often their problem may be that they are not so fluent in English” [41]
- “Non-English students struggle with dealing with the help systems. They are familiar with synonyms so unless the word is exactly the same as the one they have typed in they cannot find it.” [43]

Often instructors offered comments about the difficulties, confusion and extreme mental effort experienced by the Strivers, but did not offer a reason that this was occurring:

- “Extremely difficult for the bottom ones to understand. It’s hard for them. The bottom ones avoid it [learning from the problem].” [8]
- “Often have no idea what to do. Shows to me that they are really not able to read the problem and try to understand the components – I guess that’s “off the scale.” [11]

Several instructors commented on the additional load imposed by the complexity of the language syntax or the environment, particularly if students had not encountered them previously:

- “some get stuck on syntax - if you look at how it’s written, it won’t compile and run, so we find it hard to teach them to pinpoint errors in the code ..

they are distracted by debugging and ultimately they lose the motivation to look further.” [36]

- “the bottom 10% of students are usually unfamiliar with the programming environment including strict logic, so they have to put in a lot more effort.” [41]
- “they don’t have the [programming] language skills so its about 9, and the rest of it becomes impossible because they get stuck and they can’t go any further.” [44]
- “What we find with the students is if the environment is too complicated then they don’t know the difference between the environment and the language, and if its too simple, then it’s not giving them any help. Like the command line – it’s not a good way of teaching because they have to come to grips with the file structure, and things, so you need an environment that takes away those sort of mechanical elements but still is not full of hundreds of different features that confuse them.” [7]
- “it requires an enormous mental effort because it’s so new, both the language and the environment” [9]

Another broad observation by participants is the general lack of concept generalisation displayed. The high cognitive load experienced by these students is shown in their inability to form schema, and hence to generalise and notice patterns. No transfer is shown by these students:

- “..generalisation ... or noticing patterns, is extremely difficult for students. Plenty of times I give them identical problems which to me are identical and then the students see them as completely different problems that are unrelated to previous ones. They just don’t see it. What do they learn from that ... ?” [1]
- “They can do the same problem 4 times in a row and trip over the same bug. Very frustrating actually.” [22]
- “there are students in the bottom portion of the class who will spend a lot more time trying to work out a problem than the average student, but they won’t learn from the experience. And there are students that will.” [38]

The view that students in the bottom 10% of a programming course are simply without the capacity to learn this content, is lamented in a comment by Participant Number 19:

- “the bottom 10% just can’t do it - they flounder” [19]

The final theme offered by instructors concerned the structure of materials or the course expectations. They indicated that in some cases, the bottom 10% of students are expected to fail, and accommodations are not made for these students.

- “most coordinators running these type of courses focus too much on the stronger students: and they should focus on the weaker students.” [41]
- “the bottom 10% of students wouldn’t be expected to pass the unit anyway” [43]

- “the assignments are set as a challenge, but not something that is impossible. The good students find them easy, the poor students struggle, and that’s the way it is, we can’t have them too easy or too hard.” [7]

5.2.4 Summary

The comments offered by participants may be summarised as follows:

- many identify multiple profiles of students within the bottom 10% of a programming course;
- some students (Ghosts) are enrolled but never have any intention of attending classes, let alone learning content;
- some students (Idlers) are attending classes, but do not apply themselves to designated tasks (for various reasons) and as such, do not apply sufficient mental effort to enable learning of content;
- some students (Strivers, who were identified by the majority of participants as existing in the bottom 10% of performance within a course) are attending classes, are motivated, are completing (as best they can) designated tasks, are exerting very high to extreme levels of mental effort on everything ... and yet are not demonstrating learning.

We argue that, despite the mental effort measures being based on perceptions by instructors of their student cohort, it is important to note that when probed, many of the instructors made a clear distinction in student failure due to non-engagement with the materials versus those who failed despite their focussed application and effort to the learning resources.

The comments from participants indicate that their perceptions of the Strivers are that they are suffering from excessive mental effort (cognitive overload). Specific themes identified by participants within this profile included:

- Lack of ability in problem solving
- Lack of innate aptitude for programming
- Lack of literacy, comprehension, and analysis skills
- Lack of English due to English being a second language
- Difficulty of the computer language and/or environment being used
- Lack of capacity to generalise concepts
- Instructional materials that do not cater for their needs.

6 Discussion

The results of the current paper are disturbing.

A clear majority (75%) of participants who are academics teaching introductory computer programming courses (units) indicated explicitly the view that there are students in the "bottom 10% of performance" of these courses who are applying themselves to the set learning activities, are completing the assignments (as best they can) and yet they are failing to learn.

Moreover, the reported measures of perceived mental effort by the most frequent (modal) participant response

indicates that these very same students who are failing to learn are putting in **extreme levels** of mental effort. These students cannot be asked to do more.

If these students are to have success in learning computer programming then aspects of the instructional design for introductory programming courses (units) must change.

Many participants indicated that such changes have already occurred as demonstrated by selection of programming language and the increased use of IDEs over the period 2003 to 2010, predominately for pedagogical benefits - that is, to help students’ learning. Yet students still fail to learn, despite their best (mental) efforts.

It is likely that a range of factors contribute to the difficulty of learning computer programming in the current regime. These include the nature of programming itself, the apparent necessity to house activities within a computer language (with associated syntax) and often the inclusion of problem-solving activities.

Cognitive Load Theory warns that the limitations of human cognitive resources means that learning is prone to fail if these resources are not carefully managed in a teaching-learning transaction. The results of the current paper indicate that many students in the lower end of performance within introductory programming courses need to apply large tracts of their cognitive resources to cope with the intrinsic nature of the to-be-learnt content. These students are also required to deploy large, and possibly unnecessary whilst learning basic concepts, levels of cognitive resources to extraneous aspects of instruction, such as the nuances of a programming language and syntax. With such high levels of cognitive resources already deployed it is likely that there is, effectively, nothing left to give to the processes of learning.

Cognitive Load Theory has proven to be an extremely effective utility in engineering better instructional designs for traditionally complex and difficult areas of study such as mathematics, physics and electrical circuitry (see Sweller 1999). Possibilities of improving the instructional design of introductory programming courses may also follow by embracing cognitive analysis of the processes and dynamics associated with learning computer programming.

Perhaps all people associated with teaching introductory programming need to revisit the question "What are the objectives of an introductory programming course?". The extent to which the answer involves the gaining of core programming concepts, as opposed to skills in any particular language and syntax, may assist in better focussing upon available and effective instructional strategies.

There is also clear theoretical argument that some of the traditional strategies used, such as a focus on problem-solving, may actively block learning. In other technically complex domains such as mathematics (Sweller & Cooper 1985), engineering (Moreno et al. 2006), and psychology (Renkl et al. 2004) the benefits of using worked examples compared to problem solving as a strategy for effective teaching and learning has been demonstrated. It has also been suggested that worked examples and faded worked examples will be beneficial

for teaching introductory programming (Gray et al. 2007) though this remains to be tested.

Greater awareness and understanding of learning through the lens of Cognitive Load Theory, with particular focus on the learning dynamics of novice programmers, may aid in identifying an appropriate response to the design, development and delivery of introductory programming courses.

7 References

- Ayres, P., & Sweller, J. (2005): The split-attention principle in multimedia learning. In *The Cambridge Handbook of Multimedia Learning* 135-146. Mayer R. (Ed.), New York, Cambridge University Press.
- Chandler, P. & Sweller, J. (1992): The split-attention effect as a factor in the design of instruction. *British Journal of Educational Psychology*, **62**(2):233-246.
- Chi, M., Glaser, R. & Rees, E. (1982): Expertise in problem solving. In *Advances in the Psychology of Human Intelligence* 7-75. Sternberg R. (Ed.). Hillsdale, NJ: Erlbaum.
- Cooper, G. & Sweller, J. (1987): The effects of schema acquisition and rule automation on mathematical problem solving transfer. *Journal of Educational Psychology*, **79**: 347-362.
- Cooper, G., Tindall-Ford, S., Chandler, P. & Sweller, J. (2001): Learning by imagining. *Journal of Experimental Psychology Applied*. **7**(1):68-82.
- De Raadt, M., Watson, R. and Toleman, M. (2002): Language trends in introductory programming courses. *Proc. Informing Science and IT Education Conference*, Cork, Ireland, Cohen, E. and Boyd, E. (Eds). InformingScience.org
- De Raadt, M., Watson, R. and Toleman, M. (2004): Introductory programming: what's happening today and will there be any students to teach tomorrow? *Proceedings of the sixth conference on Australasian Computing Education*. Dunedin, New Zealand, **30**:277-284, Australian Computing Society, Inc.
- Du Boulay, B (1989): Some difficulties in learning to program. In *Studying the Novice Programmer*. 283-299. Soloway, E. & Spohrer, J.C. (Eds.). Hillsdale, NJ: Lawrence Erlbaum.
- Gray, S. et al., 2007. Suggestions for graduated exposure to programming concepts using fading worked examples. Proceedings of the third international workshop on Computing education research.
- Hunt, M. (1982). *The Universe Within*. Great Britain, The Harvester Press.
- Kirschner, P.A. (2002): Cognitive load theory: implications of cognitive load theory on the design of learning. *Learning and Instruction*. **12**(1):1-10.
- Mason, R., Cooper, G. & de Raadt, M., in press. Trends in Introductory Programming Courses in Australian Universities – Languages, Environments and Pedagogy. *Proceedings of the Fourteenth Australasian Computing Education Conference (ACE2012)*. Melbourne, Australia: Australian Computer Society, Inc.
- Mayer, R. E. (2005): Cognitive theory of multimedia learning. In *The Cambridge Handbook of Multimedia Learning*. 31-48. R. Mayer (Ed.). New York, Cambridge University Press.
- Miller, G. (1956): The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review*. **63**:81-97
- Moreno, R., Reisslein, M. & Delgoda, G., 2006. Toward a Fundamental Understanding of Worked Example Instruction: Impact of Means-Ends Practice, Backward/Forward Fading, and Adaptivity. In Proceedings of the 36th Annual Frontiers in Education Conference. San Diego, CA: IEEE, pp. 5-10.
- Paas, F., Renkl, A., & Sweller, J. (2003): Cognitive load theory and instructional design: Recent developments. *Educational Psychologist* **38**(1):1-4.
- Paas, F., Renkl, A. & Sweller, J. (2004): Cognitive Load Theory: Instructional implications of the interaction between information structures and cognitive architecture. *Instructional Science* **32**:1-8.
- Paas, F. & Van Merriënboer, J. (1993): The efficiency of instructional conditions: An approach to combine mental effort and performance measures. *Human Factors* **35**(4):737-743.
- Paas, F. & Van Merriënboer, J. (1994): Variability of worked examples and transfer of geometrical problem solving skills: A cognitive load approach. *Journal of Educational Psychology* **86**:122-133.
- Renkl, A., Atkinson, R.K. & Große, C.S., 2004. How Fading Worked Solution Steps Works – A Cognitive Load Perspective. *Instructional Science*, **32**, pp.59-82.
- Renkl, A. (2005): The worked-out example principle in multimedia learning. In *The Cambridge Handbook of Multimedia Learning* 229-245. R. Mayer (Ed.). New York, Cambridge University Press.
- Shiffrin, R. & Schneider, W. (1977): Controlled and automatic human information processing II. Perceptual learning, automatic attending and a general theory. *Psychological Review* **84**:127-190.
- Sweller, J. (1988): Cognitive load during problem solving: Effects on learning. *Cognitive Science*, **12**:257-285.
- Sweller, J. (1999): *Instructional Design in Technical Areas*. Melbourne, Australia, ACER Press.
- Sweller, J. & Chandler, P. (1994): Why some material is difficult to learn. *Cognition and Instruction* **12**(3):185-233.
- Sweller, J. & Cooper, G., 1985. The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra. *Cognition and Instruction*, **2**(1), pp.59-89.

Author Index

- Ahadi, Alireza, 77, 87
Alammary, Ali Saleh, 121
- Berglund, Anders, 103
- Carbone, Angela, iii, 61, 121, 157, 167
Ceddia, Jason, 167
Chinn, Donald, 61
Clear, Tony, 61, 103
Cooper, Graham, 33, 187
Corney, Malcolm, 53, 77, 87
Craneffield, Stephen, 43
Curran, James, 53
- D'Souza, Daryl, 53, 61
Daniels, Mats, 95
de Raadt, Michael, iii, 33, 61, 71
Dick, Martin, 177
- Fenwick, Joel, 141
Fidge, Colin, 53
Firmin, Sally: Sheard, Judy, 157
- Gluga, Richard, 53, 147
- Hamilton, Margaret, 53
Harland, James, 53
Hogan, James, 53
Hu, Minjie, 43
Hurst, John, 157
Hyland, Peter, 15
- Kölling, Michael, 3
Kay, Judy, 53, 147
- Kleitman, Sabina, 147
Koppi, Tony, 7, 25
- Laakso, Mikko-Jussi, 61
Lever, Tim, 147
Lister, Raymond, 53, 61, 77, 87, 147
- Mason, Raina, 33, 187
Mcgill, Tanya, 15
Murphy, Tara, 53
- Naghdy, Golshah, 7
- Pears, Arnold, 95
Philpot, Anne, 61
Pilgrim, Chris, 25
- Reye, James, 131
Risco, Silviu, 131
Roberts, Madeleine, 7, 15
Roggenkamp, Mike, 53
- Sheard, Judy, 53, 61, 121, 177
Simon, 53, 61
Skene, James, 61
Sutton, Peter, 141
- Teague, Donna, 53, 77, 87
Thota, Neena, 103
Trabelsi, Zouheir, 113
- Warburton, Geoff, 61
Winikoff, Michael, 43

Recent Volumes in the CRPIT Series

ISSN 1445-1336

Listed below are some of the latest volumes published in the ACS Series *Conferences in Research and Practice in Information Technology*. The full text of most papers (in either PDF or Postscript format) is available at the series website <http://crpit.com>.

- Volume 113 - Computer Science 2011**
Edited by Mark Reynolds, The University of Western Australia, Australia. January 2011. 978-1-920682-93-4. Contains the proceedings of the Thirty-Fourth Australasian Computer Science Conference (ACSC 2011), Perth, Australia, 17-20 January 2011.
- Volume 114 - Computing Education 2011**
Edited by John Hamer, University of Auckland, New Zealand and Michael de Raadt, University of Southern Queensland, Australia. January 2011. 978-1-920682-94-1. Contains the proceedings of the Thirteenth Australasian Computing Education Conference (ACE 2011), Perth, Australia, 17-20 January 2011.
- Volume 115 - Database Technologies 2011**
Edited by Heng Tao Shen, The University of Queensland, Australia and Yanchun Zhang, Victoria University, Australia. January 2011. 978-1-920682-95-8. Contains the proceedings of the Twenty-Second Australasian Database Conference (ADC 2011), Perth, Australia, 17-20 January 2011.
- Volume 116 - Information Security 2011**
Edited by Colin Boyd, Queensland University of Technology, Australia and Josef Pieprzyk, Macquarie University, Australia. January 2011. 978-1-920682-96-5. Contains the proceedings of the Ninth Australasian Information Security Conference (AISC 2011), Perth, Australia, 17-20 January 2011.
- Volume 117 - User Interfaces 2011**
Edited by Christof Lutteroth, University of Auckland, New Zealand and Haifeng Shen, Flinders University, Australia. January 2011. 978-1-920682-97-2. Contains the proceedings of the Twelfth Australasian User Interface Conference (AUIC2011), Perth, Australia, 17-20 January 2011.
- Volume 118 - Parallel and Distributed Computing 2011**
Edited by Jinjun Chen, Swinburne University of Technology, Australia and Rajiv Ranjan, University of New South Wales, Australia. January 2011. 978-1-920682-98-9. Contains the proceedings of the Ninth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia, 17-20 January 2011.
- Volume 119 - Theory of Computing 2011**
Edited by Alex Potanin, Victoria University of Wellington, New Zealand and Taso Viglas, University of Sydney, Australia. January 2011. 978-1-920682-99-6. Contains the proceedings of the Seventeenth Computing: The Australasian Theory Symposium (CATS 2011), Perth, Australia, 17-20 January 2011.
- Volume 120 - Health Informatics and Knowledge Management 2011**
Edited by Kerryn Butler-Henderson, Curtin University, Australia and Tony Sahama, Queensland University of Technology, Australia. January 2011. 978-1-921770-00-5. Contains the proceedings of the Fifth Australasian Workshop on Health Informatics and Knowledge Management (HIKM 2011), Perth, Australia, 17-20 January 2011.
- Volume 121 - Data Mining and Analytics 2011**
Edited by Peter Vamplew, University of Ballarat, Australia, Andrew Stranieri, University of Ballarat, Australia, Kok-Leong Ong, Deakin University, Australia, Peter Christen, Australian National University, Australia and Paul J. Kennedy, University of Technology, Sydney, Australia. December 2011. 978-1-921770-02-9. Contains the proceedings of the Ninth Australasian Data Mining Conference (AusDM'11), Ballarat, Australia, 1-2 December 2011.
- Volume 122 - Computer Science 2012**
Edited by Mark Reynolds, The University of Western Australia, Australia and Bruce Thomas, University of South Australia. January 2012. 978-1-921770-03-6. Contains the proceedings of the Thirty-Fifth Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, 30 January - 3 February 2012.
- Volume 123 - Computing Education 2012**
Edited by Michael de Raadt, Moodle Pty Ltd and Angela Carbone, Monash University, Australia. January 2012. 978-1-921770-04-3. Contains the proceedings of the Fourteenth Australasian Computing Education Conference (ACE 2012), Melbourne, Australia, 30 January - 3 February 2012.
- Volume 124 - Database Technologies 2012**
Edited by Rui Zhang, The University of Melbourne, Australia and Yanchun Zhang, Victoria University, Australia. January 2012. 978-1-920682-95-8. Contains the proceedings of the Twenty-Third Australasian Database Conference (ADC 2012), Melbourne, Australia, 30 January - 3 February 2012.
- Volume 125 - Information Security 2012**
Edited by Josef Pieprzyk, Macquarie University, Australia and Clark Thomborson, The University of Auckland, New Zealand. January 2012. 978-1-921770-06-7. Contains the proceedings of the Tenth Australasian Information Security Conference (AISC 2012), Melbourne, Australia, 30 January - 3 February 2012.
- Volume 126 - User Interfaces 2012**
Edited by Haifeng Shen, Flinders University, Australia and Ross T. Smith, University of South Australia, Australia. January 2012. 978-1-921770-07-4. Contains the proceedings of the Thirteenth Australasian User Interface Conference (AUIC2012), Melbourne, Australia, 30 January - 3 February 2012.
- Volume 127 - Parallel and Distributed Computing 2012**
Edited by Jinjun Chen, University of Technology, Sydney, Australia and Rajiv Ranjan, CSIRO ICT Centre, Australia. January 2012. 978-1-921770-08-1. Contains the proceedings of the Tenth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012), Melbourne, Australia, 30 January - 3 February 2012.
- Volume 128 - Theory of Computing 2012**
Edited by Julián Mestre, University of Sydney, Australia. January 2012. 978-1-921770-09-8. Contains the proceedings of the Eighteenth Computing: The Australasian Theory Symposium (CATS 2012), Melbourne, Australia, 30 January - 3 February 2012.
- Volume 129 - Health Informatics and Knowledge Management 2012**
Edited by Kerryn Butler-Henderson, Curtin University, Australia and Kathleen Gray, University of Melbourne, Australia. January 2012. 978-1-921770-10-4. Contains the proceedings of the Fifth Australasian Workshop on Health Informatics and Knowledge Management (HIKM 2012), Melbourne, Australia, 30 January - 3 February 2012.
- Volume 130 - Conceptual Modelling 2012**
Edited by Aditya Ghose, University of Wollongong, Australia and Flavio Ferrarotti, Victoria University of Wellington, New Zealand. January 2012. 978-1-921770-11-1. Contains the proceedings of the Eighth Asia-Pacific Conference on Conceptual Modelling (APCCM 2012), Melbourne, Australia, 31 January - 3 February 2012.
- Volume 131 - Advances in Ontologies 2010**
Edited by Thomas Meyer, UKZN/CSIR Meraka Centre for Artificial Intelligence Research, South Africa, Mehmet Orgun, Macquarie University, Australia and Kerry Taylor, CSIRO ICT Centre, Australia. December 2010. 978-1-921770-00-5. Contains the proceedings of the Sixth Australasian Ontology Workshop 2010 (AOW 2010), Adelaide, Australia, 7th December 2010.