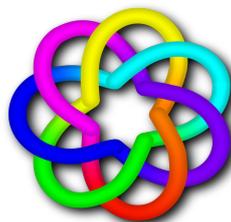


CONFERENCES IN RESEARCH AND PRACTICE IN
INFORMATION TECHNOLOGY

VOLUME 136

COMPUTING EDUCATION 2013

AUSTRALIAN COMPUTER SCIENCE COMMUNICATIONS, VOLUME 35, NUMBER 2



COMPUTING EDUCATION 2013

Proceedings of the
Fifteenth Australasian Computing Education Conference
(ACE 2013), Adelaide, Australia,
29 January – 1 February 2013

Angela Carbone and Jacqueline Whalley, Eds.

Volume 136 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 2013. Proceedings of the Fifteenth Australasian Computing Education Conference (ACE 2013), Adelaide, Australia, 29 January – 1 February 2013

Conferences in Research and Practice in Information Technology, Volume 136.

Copyright ©2013, 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:

Angela Carbone

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

Jacqueline Whalley

School of Computing and Mathematical Sciences
AUT University Auckland
New Zealand
Email: jacqueline.whalley@aut.ac.nz

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 136.
ISSN 1445-1336.
ISBN 978-1-921770-21-0.

Document engineering, January 2014 by CRPIT
On-line proceedings, January 2014 by the University of Western Sydney
Electronic media production, January 2014 by the University of South Australia

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 Fifteenth Australasian Computing Education Conference (ACE 2013), Adelaide, Australia, 29 January – 1 February 2013

Preface	vii
Programme Committee	viii
Organising Committee	ix
Welcome from the Organising Committee	x
CORE - Computing Research & Education	xi
ACSW Conferences and the Australian Computer Science Communications	xii
ACSW and ACE 2013 Sponsors	xiv

Contributed Papers

A Process for Novice Programming Using Goals and Plans	3
<i>Minjie Hu, Michael Winikoff and Stephen Cranefield</i>	
Its Never Too Early: Pair Programming in CS1	13
<i>Krissi Wood, Dale Parsons, Joy Gasson and Patricia Haden</i>	
Distractions in Programming Environments	23
<i>Raina Mason and Graham Cooper</i>	
Identifying career outcomes as the first step in ICT curricula development	31
<i>Nicole Herbert, Kristy de Salas, Ian Lewis, Michael Cameron-Jones, Winyu Chinthammit, Julian Dermoudy, Leonie Ellis and Matthew Springer</i>	
Student Concerns in Introductory Programming Courses	41
<i>Angela Carbone, Jason Ceddia, Simon, Daryl D'Souza and Raina Mason</i>	
Stakeholder-Led Curriculum Design	51
<i>Nicole Herbert, Julian Dermoudy, Leonie Ellis, Michael Cameron-Jones, Winyu Chinthammit, Kristy De Salas, Ian Lewis and Matthew Springer</i>	
Measuring the difficulty of code comprehension tasks using software metrics	59
<i>Nadia Kasto and Jacqueline Whalley</i>	
Revisiting models of human conceptualisation in the context of a programming examination	67
<i>Jacqueline Whalley and Nadia Kasto</i>	
A conceptual model for reflecting on expected learning vs. demonstrated student performance	77
<i>Richard Gluga, Judy Kay, Raymond Lister, Simon, Michael Charleston and Donna Teague</i>	
A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers	87
<i>Donna Teague, Malcolm Corney, Alireza Ahadi and Raymond Lister</i>	
What vs. How: Comparing Students' Testing and Coding Skills	97
<i>Colin Fidge, James Hogan and Raymond Lister</i>	

Visualisation of Learning Management System Usage for Early Detection of Students At Risk of Failure in Computer Science Courses	107
<i>Thomas Haig, Katrina Falkner and Nickolas Falkner</i>	
A Comparative Analysis of Results on Programming Exams	117
<i>James Harland, Daryl D'Souza and Margaret Hamilton</i>	
Examining Student Reflections from a Constructively Aligned Introductory Programming Unit	127
<i>Andrew Cain and Clinton Woodward</i>	
Computational Thinking and Practice A Generic Approach to Computing in Danish High Schools ..	137
<i>Michael E. Caspersen and Palle Nowack</i>	
How difficult are exams? A framework for assessing the complexity of introductory programming exams	145
<i>Judy Sheard, Simon, Angela Carbone, Donald Chinn, Tony Clear, Malcolm Corney, Daryl D'Souza, Joel Fenwick, James Harland, Mikko-Jussi Laakso and Donna Teague</i>	
Integrating Source Code Plagiarism into a Virtual Learning Environment: Benefits for Students and Staff	155
<i>Tri Le Nguyen, Angela Carbone, Judy Sheard and Margot Schuhmacher</i>	
Author Index	165

Preface

Welcome to the Fifteenth Australasian Computing Education Conference (ACE2013). This year the ACE2013 conference, which is part of the Australasian Computer Science Week, is being held at the University of South Australia, Adelaide, Australia from 29 January to 1 February, 2013.

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 17 full papers from the 37 papers submitted, giving an acceptance rate of 49%. The number of submissions was slightly less than the 43 papers submitted in the previous year, however this year we had ten papers submitted by research students, which reflects the growing research interest in computing education. We again see a strong national and international presence, with submissions from Australia, New Zealand, Finland, United States, Malaysia, Brazil and Denmark. A variety of topics are presented in this year's papers, including: novice programming; assessment, curricula design and pedagogy; and student learning. Many of the papers present new innovations and demonstrate high quality research.

This year we invited Dr Mats Daniels, Senior academic from the Department of Information Technology, Uppsala University, Sweden to deliver an invited address titled *Taking Competencies Seriously*. He will also take part as a panel member in the first ACE doctoral consortium, sponsored by the Australian Council of Deans ICT Learning and Teaching Academy (ALTA). ALTA covered the ACE registration fee for twelve PhD students to discuss and explore their research interests and career objectives with a panel of established researchers in computing education research. The doctoral consortium is chaired by Dr Margaret Hamilton from RMIT University with discussants including: Jacqueline Whalley and Tony Clear (AUT University), Daryl DSouza and James Harland (RMIT) and Angela Carbone (Monash University).

As with past ACE conferences, we are continuing to hold workshops. Three workshops have been organized, these include: *Model-driven programming education* led by Michael Caspersen; *Developing teamwork that works* supported by Australian Council of Deans ICT (ACD ICT) Teaching Fellowship led by Elena Sitnikova, Patricia Kelly and Diana Collett; *Writing a good exam for a programming course* led by Simon, Judy Sheard, Angela Carbone, Malcolm Corney, Raymond Lister and Donna Teague.

This year ACE awarded a best paper and best student paper. The best paper was awarded to:

- ★ Distractions in Programming Environments
Raina Mason and Graham Cooper

Two other papers were also highly commendable from the reviews:

- ★ Examining Student Reflections from a Constructively Aligned Introductory Programming Unit
Andrew Cain and Clinton Woodward
- ★ How difficult are exams? A framework for assessing the complexity of introductory programming exams
Judy Sheard, Simon, Angela Carbone, Donald Chinn, Tony Clear, Malcolm Corney, Daryl D'Souza, Joel Fenwick, James Harland, Mikko-Jussi Laakso and Donna Teague

The best student paper was awarded to:

- ★ A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers
Donna Teague, Malcolm Corney, Alireza Ahadi and Raymond Lister

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 its proceedings publication possible, and we thank CORE, ALTA, our hosts University of South Australia, Adelaide, and the Australasian Computing Education executive for the opportunity to chair the ACE2013 conference.

Angela Carbone
Monash University

Jacqueline Whalley
AUT University Auckland

ACE 2013 Conference Co-chairs
January 2013

Programme Committee and Additional Referees

Chairs

Angela Carbone, Monash University, Australia
Jacqueline Whalley, AUT University, New Zealand

Members

David J. Barnes, University of Kent, UK
Tim Bell, University of Canterbury, New Zealand
Jason Ceddia, Monash University, Australia
Alison Clear, AUT University, New Zealand
Tony Clear, AUT University, New Zealand
Mats Daniels, Uppsala University, Sweden
Paul Denny, The University of Auckland, New Zealand
Michael de Raadt, Moodle, Australia
Julian Dermoudy, University of Tasmania, Australia
John Hamer, University of Auckland, New Zealand
Margaret Hamilton, RMIT University, Australia
Chris Johnson Australian National University, Australia
Michael Kolling, University of Kent, UK
Mikko Laakso, University of Turku, Finland
Andrew Luxton-Reilly, University of Auckland, New Zealand
Raymond Lister, University of Technology, Sydney, Australia
Chris McDonald, University of Western Australia, Australia
Dale Parsons, Otago Polytechnic, New Zealand
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
Daryl DSouza, RMIT University, Australia
Simon, University of Newcastle, Australia
Josh Tenenberg, University of Washington, USA
Errol Thompson, Aston University, United Kingdom

Conference Webmaster

Jason Ceddia, Monash University, Australia

Organising Committee

Chair

Dr. Ivan Lee

Finance Chair

Dr. Wolfgang Mayer

Publication Chair

Dr. Raymond Choo

Local Arrangement Chair

Dr. Grant Wigley

Registration Chair

Dr. Jinhai Cai

Welcome from the Organising Committee

On behalf of the Organising Committee, it is our pleasure to welcome you to Adelaide and to the 2013 Australasian Computer Science Week (ACSW 2013). Adelaide is the capital city of South Australia, and it is one of the most liveable cities in the world. ACSW 2013 will be hosted in the City West Campus of University of South Australia (UniSA), which is situated at the north-west corner of the Adelaide city centre.

ACSW is the premier event for Computer Science researchers in Australasia. ACSW2013 consists of conferences covering a wide range of topics in Computer Science and related area, including:

- Australasian Computer Science Conference (ACSC) (Chaired by Bruce Thomas)
- Australasian Database Conference (ADC) (Chaired by Hua Wang and Rui Zhang)
- Australasian Computing Education Conference (ACE) (Chaired by Angela Carbone and Jacqueline Whalley)
- Australasian Information Security Conference (AISC) (Chaired by Clark Thomborson and Udaya Parampalli)
- Australasian User Interface Conference (AUIC) (Chaired by Ross T. Smith and Burkhard C. Wünsche)
- Computing: Australasian Theory Symposium (CATS) (Chaired by Tony Wirth)
- Australasian Symposium on Parallel and Distributed Computing (AusPDC) (Chaired by Bahman Javadi and Saurabh Kumar Garg)
- Australasian Workshop on Health Informatics and Knowledge Management (HIKM) (Chaired by Kathleen Gray and Andy Koronios)
- Asia-Pacific Conference on Conceptual Modelling (APCCM) (Chaired by Flavio Ferrarotti and Georg Grossmann)
- Australasian Web Conference (AWC2013) (Chaired by Helen Ashman, Michael Sheng and Andrew Trotman)

In addition to the technical program, we also put together social activities for further interactions among our participants. A welcome reception will be held at Rockford Hotel's Rooftop Pool area, to enjoy the fresh air and panoramic views of the cityscape during Adelaide's dry summer season. The conference banquet will be held in Adelaide Convention Centre's Panorama Suite, to experience an expansive view of Adelaide's serene riverside parklands through the suite's seamless floor to ceiling windows.

Organising a conference is an enormous amount of work even with many hands and a very smooth cooperation, and this year has been no exception. We would like to share with you our gratitude towards all members of the organising committee for their dedication to the success of ACSW2013. Working like one person for a common goal in the demanding task of ACSW organisation made us proud that we got involved in this effort. We also thank all conference co-chairs and reviewers, for putting together conference programs which is the heart of ACSW. Special thanks goes to Alex Potanin, who shared valuable experiences in organising ACSW and provided endless help as the steering committee chair. We'd also like to thank Elyse Perin from UniSA, for her true dedication and tireless work in conference registration and event organisation. Last, but not least, we would like to thank all speakers and attendees, and we look forward to several stimulating discussions.

We hope your stay here will be both rewarding and memorable.

Ivan Lee

School of Information Technology & Mathematical Sciences

ACSW2013 General Chair

January, 2013

CORE - Computing Research & Education

CORE welcomes all delegates to ACSW2013 in Adelaide. 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 eight other events - ACE, AISC, AUIC, AusPDC, HIKM, ACDC, APCCM and AWC which build on the diversity of the Australasian computing community.

In 2013, we have again chosen to feature a small number of keynote speakers from across the discipline: Riccardo Bellazzi (HIKM), and Divyakant Agrawal (ADC), Maki Sugimoto (AUIC), and Wen Gao. I thank them for their contributions to ACSW2013. I also thank invited speakers in some of the individual conferences, and the CORE award winner Michael Sheng (CORE Chris Wallace 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 Ivan Lee and his colleagues for organising what promises to be a strong event.

The past year has been turbulent for our disciplines. ERA2012 included conferences as we had pushed for, but as a peer review discipline. This turned out to be good for our disciplines, with many more Universities being assessed and an overall improvement in the visibility of research in our disciplines. The next step must be to improve our relative success rates in ARC grant schemes, the most likely hypothesis for our low rates of success is how harshly we assess each others' proposals, a phenomenon which demonstrably occurs in the US NFS. As a US Head of Dept explained to me, "in CS we circle the wagons and shoot within".

Beyond research issues, in 2013 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? In the US there has been massive growth in undergraduate CS numbers of 25 to 40% in many places, which we should aim to replicate. ACSW will feature a joint CORE, ACDICT, NICTA and ACS discussion on ICT Skills, which will inform our future directions.

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 2012; in particular, I thank Alex Potanin, Alan Fekete, Aditya Ghose, Justin Zobel, John Grundy, 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.

I am standing down as President at this ACSW. I have enjoyed the role, and am pleased to have had some positive impact on ERA2012 during my time. Thank you all for the opportunity to represent you for the last 3 years.

Tom Gedeon

President, CORE
January, 2013

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.

2014. Volume 36. Host and Venue - AUT University, Auckland, New Zealand.

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
APCCM	Asia-Pacific Conference on Conceptual Modelling
AUIC	Australasian User Interface Conference
AusPDC	Australasian Symposium on Parallel and Distributed Computing (replaces AusGrid)
AWC	Australasian Web Conference
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 2013 Sponsors

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



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



Australian Computer Society,
www.acs.org.au



University of
South Australia

University of South Australia,
www.unisa.edu.au/



Monash University,
www.monash.edu.au



AUT University,
www.aut.ac.nz



Association for
Computing Machinery

Advancing Computing as a Science & Profession

Association for Computing Machinery,
www.acm.org



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



Australian Council of Deans of Information and
Communications Technology,
www.acdict.edu.au

CONTRIBUTED PAPERS

A Process for Novice Programming Using Goals and Plans

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

scrانefield@infoscience.
otago.ac.nz

Abstract

We propose to improve the teaching of programming to novices by using a clearly-defined and detailed process that makes use of goals and plans, and a visual programming language. We present a simple notation for designing programs in terms of data flow networks of goals and plans, and define a detailed process that uses this notation, and that ultimately results in a program in a visual programming language (BYOB). Results from an evaluation are presented that show the effectiveness of this approach.

Keywords: Goal, Plan, Process of Programming.

1 Introduction

A range of studies across institutions and countries have observed that novices struggle in introductory programming (Lister et al. 2004, McCracken et al. 2001). Accordingly a wide range of approaches have been proposed to improve novices' learning of programming. For example, the problem-solving approach emphasises the development of problem-solving skills connected to programming (Pears et al. 2007); problem-based learning (PBL) is based on solving a "large real-world" problem collaboratively in groups (Kay et al. 2000); collaborative learning provides support and enhances communication in learning environments to promote students' high level cognitive skills (Rößling et al. 2008); psychological analysis considers the mental models of novices, and proposes various concrete conceptual models to help novices to understand programming (Mayer 1981, Winslow 1996); programming visualisation provides visual support towards the development of viable mental models and engages novices in an active learning activity to improve their understanding of programming and help their learning (Ben-Ari 2001, Naps et al. 2003); and game programming attracts, motivates and engages novices to learn programming by using computer games as a subject based on multimedia, pre-developed libraries or micro-worlds (Guzdial and Soloway 2002, Kölling and Henriksen 2005).

Copyright © 2013, Australian Computer Society, Inc. This paper appeared at the 15th Australasian Computer Education Conference (ACE 2013), Adelaide, South Australia, January-February 2013. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 136. A. Carbone and J. Whalley, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

However, these approaches have not been widely adopted. Although some approaches (e.g. PBL) have been demonstrated to be highly effective (Kay et al. 2000), they are quite costly to introduce. By contrast, we aim to develop an approach for teaching novice programming that is both effective and cheap to introduce. Specifically, we propose to combine three aspects: the use of a **visual programming language**; the use of **goals and plans**; and the employment of a clear well-defined **process with feedback**.

Recently, visual programming languages (VPLs), such as Scratch and Alice, have been used to teach novice programming. Programs are built by dragging and dropping statement blocks, which helps to prevent syntax errors and enables students to make better early progress (Lister 2011). The philosophy of using VPLs to teach novices is to "let them play first, let them achieve something, ... and then sneak the explanations in" (Utting et al. 2010, p7). However, there are concerns that students might "simply mess around and never focus towards any goal" (Utting et al. 2010, p4). In other words, students may learn to program by trial and error, rather than by following a systematic approach. Therefore, they need guidance, that is, a process, for how to program.

In order to give guidance to novice programmers, we take as our point of departure the work on goals and plans. A goal is a certain objective that a program must achieve in order to solve a problem (Letovsky and Soloway 1986), and a plan (Spohrer, Soloway, and Pope 1985) corresponds to a fragment of code that performs actions to achieve a goal. In the 1980s, Soloway (1986) and his colleagues (Letovsky and Soloway 1986, Spohrer, Soloway, and Pope 1985) discovered that experts have strategies to solve problems using their own libraries of plans. They advocated structuring these libraries in terms of goals and plans, and teaching strategies for using these libraries. Subsequently, educators have been attempting to introduce goals and plans as a means of structuring the development of programs (de Raadt 2008, de Raadt et al 2006, Guzdial et al. 1998, Soloway 1986). For example, various template-based approaches for using program fragments were proposed such as "pedagogical programming patterns" (Porter and Calder 2003) and "programming strategies" (de Raadt 2008). However, this body of work did not provide a detailed process for using goals and plans in program development.

For the reasons outlined above, there is therefore an opportunity and need to provide a detailed step-by-step

process for programming by novices (Caspersen and Kölling 2009). There have been a number of approaches that focus on teaching novices a process for programming. Pattis (1990) proposed that the programming process be broken down into a series of well-defined steps, and that it is important to provide feedback from each step. Providing feedback at each step was considered to be critical in giving students confidence. In fact, feedback is at the heart of test-driven development, and Janzen and Saiedian (2006) recently proposed to improve teaching by using “test-driven learning”. However, none of these papers provided a detailed process that could be taught to novices.

A number of detailed processes for teaching novices have been proposed. For example, Programming by Numbers (Glaser, Hartel, and Garratt 2000) and TeachScheme (Felleisen et al. 2004) both provide a clear process for creating the smallest components of functions, using stepwise refinement. Both approaches are data-driven and more suited to functional programming languages than to mainstream procedural languages. Another process that has been proposed is STREAM (Caspersen and Kölling 2009), which aims to teach novices a process for object-oriented programming. However, none of these proposed processes included the use of goals and plans (or of a VPL).

Our research therefore focuses on the development of a programming process for novice programmers using goals and plans in a visual programming environment. Our programming process aims to be clearly defined, detailed, iterative, incremental, and to provide feedback at every step. Our previous work (Anonymous 2012) presented an overview of our approach, whereas this paper provides a more detailed presentation of the process.

2 Design Notation

Before providing a process for program development using goals and plans, we first need to define a design notation, which is used to capture the results of the first step of the process.

Soloway and colleagues proposed that novices develop a structure for their program in terms of a tree of goals. This tree is progressively refined, and ultimately, goals are realized as combinations of plans. Initially, Soloway (1986) proposed three ways of combining plans: sequential (Plan B begins after Plan A finishes), nested (Plan B is used as one of the steps within Plan A), and interleaved (the steps of Plan A and B are merged with each other). Subsequently Ebhrahimi (1992) proposed an additional plan combination: branching (Plan A uses either Plan B or Plan C, depending on a condition).

The term “interleaved” plans above does not indicate plans that have been interleaved, but rather plans that will need to have their implementations merged to form an executable single-threaded program. For example, consider computing an average of a sequence of numbers, which has been designed using a plan for computing the sum of the sequence, and another plan for computing the count of the sequence (number of items). An executable (procedural) program needs to read each input, and then process it, including updating both the sum and the count, before the next input value is read. For example, see the

right part of Figure 7, which shows an executable procedural program (in BYOB¹) where three plans (Input, Sum, and Count) are interleaved.

However, by adopting a data flow based representation, we can avoid the need for merging before any execution can be done. We model plans as consuming and producing sequences of data. For the above example, if we have a data flow buffer between plans, then we can execute the Input plan, store the results in a buffer, execute the Sum plan to completion, then execute the Count plan to completion, and finally proceed to compute the average by dividing the sum by the count. In other words, **by using a data flow model, we can execute unmerged plans**. This is a significant difference between control flow and data flow models when using goals and plans, because it enables novices to receive feedback before plans are merged into a final (procedural) program.

We therefore propose a notation where goals and plans are represented by icons, and are linked by arrows (denoting data flows between them). Goals can be categorized into three types: input, output, or processing (Figure 1). A simple program might have only one goal of each type and achieve these goals in sequence, but more typically a program would have multiple processing goals.



Figure 1: Example of three basic goals

A data flow is a sequence of values that “flow” between goals. Each data flow is represented by an arrow that links two goals. In the typical case, a data flow has a single source and single destination, and links two goals. It is also possible for a data flow to have two destinations, indicated with a “fork” notation (e.g. Figure 4). We make the assumption that each goal has a single out port and we use arrows to show the direction of the flow. This means that we do not need to have named ports on goal diagrams, which simplifies the initial design stage for students.

The second design stage is to produce a plan diagram corresponding to the goal diagram. Since a plan is a code segment that accomplishes a programming goal, it is visualised as a box with double lines on both sides like a sub-program icon in flowchart notation. Plan icons are used to replace all goal icons in the goal diagram, yielding a plan diagram (Figure 2). At this stage, plan ports are added and named. A data flow can be accessed from within plans by two ports: from a source (“out”) port of one plan to a destination (“in”) port of another plan. Ports are identified and named by the combination of the name of the plan and the function of the port. For example, the port name Input:out represents the out port of an Input Plan.

¹ Build Your Own Blocks (BYOB) is a variant of Scratch (see <http://scratch.mit.edu>). BYOB permits users to build new “blocks”. Each new block can include a procedure. See <http://byob.berkeley.edu6>

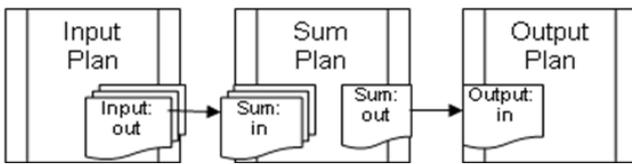


Figure 2: An example of a plan network diagram

We distinguish between ports that are associated with a data flow that only involves a single value, and those that are the end points of a data flow with a sequence of values. For example, the first data flow in Figure 2 links the port Input:out to the port Sum:in and can contain a sequence of values. However, the data flow from Sum:out to Output:in will only contain a single value.

3 Programming Process

The process of programming that we propose consists of five steps: (1) analysing goals and plans; (2) mapping the plan network to BYOB using plan blocks (where each plan is mapped to a BYOB plan block); (3) expanding plan blocks; (4) merging the expanded plan details; and (5) simplifying the merged details (Figure 3). Each step includes three or four sub-steps.

In order to provide early and frequent feedback, we adopted the test-driven learning approach. Before tackling the first step, students have a “step zero” (not shown in Figure 3) in which they study the question, and specify test cases (both input and correct output). For example, if the problem is to compute the average of input numbers, then a student might define the following two test cases: given 1, 2, and 3 the average is 2; and given 2, 3, 7 and 8 it is 5.

After each step, the student checks that the results are correct, i.e. that the design produces the expected answers for the test cases. For the first step, this checking is done by a manual desk-check, but for the following four steps, all the test results come from executable programs.

The programming process that we describe below guides the student through an incremental development process that proceeds from goal and plan concepts, to plan block design, to intermediate program, and to final code. The process is illustrated using the following example, which was originally used by Soloway (1986) to analyse goals and plans:

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

3.1 Analysing Goals and Plans

Goal analysis starts by identifying what goals the program needs to achieve. Typically a program includes at least one input, one output, and some number of processing goals, some of which may need hierarchical refinement. For the above example, the first goal is to input the values. Following this, a “compute average” goal was initially required. However, the average goal needs the sum and count of the input. Hence, the “compute average” goal can be decomposed into three goals, sum, count, and divide, where both sum and count receive the same data flow from the input goal (in parallel), and send their results to the divide goal (also in parallel). Finally, the result of the divide goal is sent to

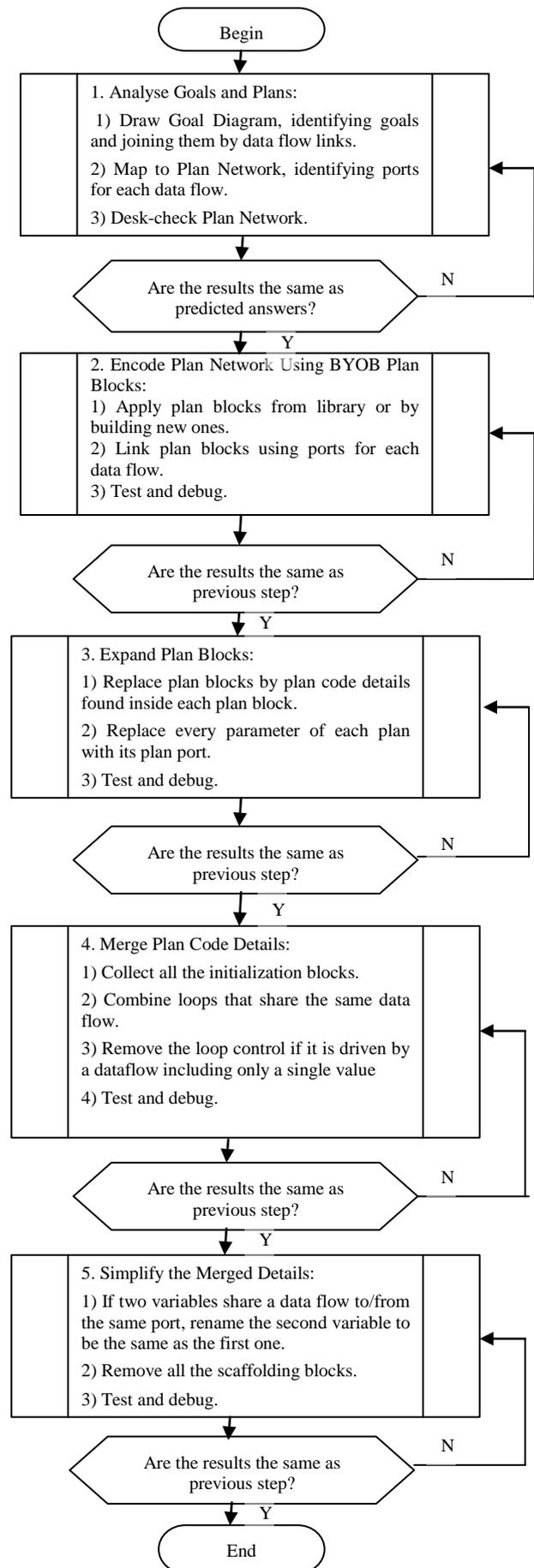


Figure 3: The process of programming

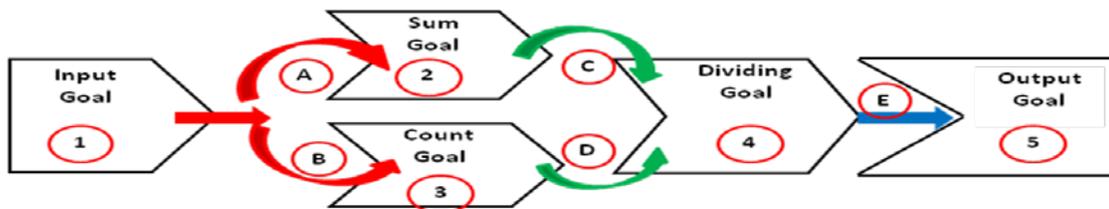


Figure 4: Example of goal

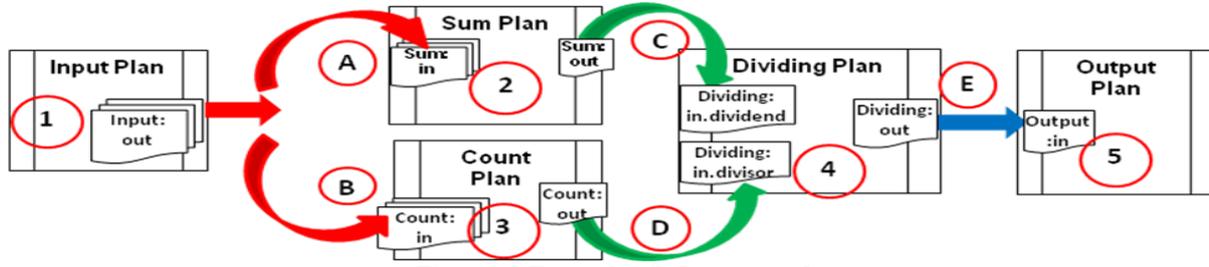


Figure 5: Example of plan network

the last goal (to output the result). Therefore, five goals (1–5) of three types are identified and presented using visual notation in Figure 4. The relationships between these goals were also identified and the goals are linked by five data flows (A–E in Figure 4). Note that the numbers 1–5 and the letters A–E are only added here for ease of explanation, i.e. they are not part of the notation.

In general, the development of the goal diagram is an incremental refinement process, in which the processing goals are hierarchically decomposed. Once goals are refined to a level where they are sufficiently fine-grained, they can be mapped to plans in a one-to-one manner (i.e. each goal becomes a plan), resulting in a plan network diagram (Figure 5). For teaching novice programming, a sufficiently fine-grained decomposition of goals means that the decomposed goals correspond to BYOB plan blocks in a provided plan library or that they can be implemented simply. For the example in Figures 4 and 5, five plan icons have been used to replace the five goals.

The next step is to realise the data flows by defining ports. In simple cases, a port name can be identified by the combination of plan name and type of the port (“in” or “out”). For example, the port on the left² of “Sum Plan” is named Sum:in; and the port on the right is named Sum:out. In cases where a plan has multiple incoming data flows, its graphical representation shows multiple in ports with different names. For example, the two in ports of the Dividing Plan are identified as dividing:in.dividend and dividing:in.divisor.

After mapping from the initial goal diagram to a plan network, a desk-check table is used to test whether or not the first step of analysis is correct (see Table 1). The table consists of two parts. The first part contains the first two columns of the table: Test Cases and Predicted Answers. The contents of these two columns are based on the test cases specified in step zero: the test case cells are the input of the test case, and the predicted answer is the expected output. The second part of the table comprises the rest of the columns, and is based on the plan network. Each column represents one port within the plan network. The cell under the “in” column of a plan is filled with a

copy of the data from the relevant “out” column, i.e. the “out” port that is linked to it by a data flow. For example, the “in” column for the Sum Plan is simply a copy of the “out” column of the Input Plan, since Input’s out port is linked to the Sum’s in port. The cells under the “out” column of a plan are filled by computing the corresponding output of a plan, given its input, for example the “out” column for Sum Plan is the sum of its inputs. Hence, the second part of the table records the data flow through the plan network.

Test Cases:		1, 2, 3, -1	2, 3, 7, 8, -1
Predicted Answers:		2	5
Input	out	1, 2, 3	2, 3, 7, 8
Sum	in	1, 2, 3	2, 3, 7, 8
	out	6	20
Count	in	1, 2, 3	2, 3, 7, 8
	out	3	4
Dividing	in.dividend	6	20
	in. divisor	3	4
	out	2	5
Output	in	2	5

Table 1: Example of a desk-check for data flow in plan ports

The first step of analysis is completed after the outputs from the last column are the same as the predicted answers in the second column for every test case. Otherwise, the analysis has to be corrected. The plan network produced is used in the next step, where it is mapped to BYOB, using plan blocks.

3.2 Encoding Plan Network Using BYOB Plan Blocks

Following the confirmation of the correctness of the goal and plan analysis by desk-checking, the diagram of the plan network (Figure 5) can be mapped to an executable plan network (Figure 6). The process for doing this is fairly straightforward and mechanical. Each plan icon is replaced by a plan block in BYOB, and every data flow is mapped to a “scaffolding block” (Link <<out port name>> to <<in port name>>) to link an out port to an in port. Note that the order of the plan blocks (1–5 in Figure

² We use a convention where in ports are on the left of a plan and out ports are on the right.

6) does matter: the plan blocks need to follow the order of the arrows in the plan network (Figure 5). For example, in this case the Input plan must be first, followed by the Sum and Count plans (in either order), and then the Dividing plan and finally the Output plan.

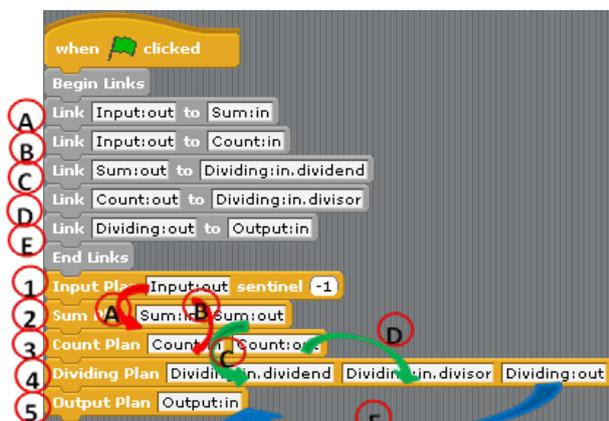


Figure 6: Example of an executable plan network in BYOB

We map plan icons to plan blocks by considering the diagram of the plan network from the previous step. Using BYOB, each plan can be implemented as a plan block with arguments that are ports to receive and/or send data flow. Processing plans (i.e. plans other than an Input or Output plan), encapsulate a procedure to receive data flow from their in ports, to process the data flow, and then send the results to their out port.

Individual plan blocks are identified from our plan library developed in BYOB. If a plan block does not exist, then the student must build it based on similar pattern of existing plan blocks in the library³. For example, considering the plan network in Figure 5, plan blocks 1–3 and 5 can be found in the provided library, but plan block 4 (“Dividing Plan”) is not in the provided library. However, the library has a “Multiplying Plan” block, which is similar and can be used as a template for developing the “Dividing Plan” block.

In order to represent a plan network in BYOB we use a number of scaffolding blocks. Note that eventually all the scaffolding blocks will be removed from the final program. There are three data flow scaffolding blocks which are used to deal with data flow within a plan. They are named “NO MORE DATA? <<port>>”, “GET DATA <<port>>”, and “SEND DATA <<datum>> <<port>>”. The first is used to find out if there is any datum in the input port of the current plan. The second is used to get a datum from the input port. The last (SEND DATA) is used to send a result of the current plan to its output port.

There are also linkage scaffolding blocks which are used to define the linkages between plan blocks. These are placed at the start, between “Begin Links” and “End Links” blocks. Each Link block specifies a linkage from an out port (source) to an in port (destination). If the current plan output port is linked by a scaffolding block

to an input port of another plan, this data flow will be sent to the linked plan through its input port. Each link in the plan network diagram (Figure 5) is directly mapped to a Link block in BYOB (Figure 6, indicated with letters A–E). For example, data flow “A” is mapped to the first two Link blocks. The first Link block links the out port of the Input Plan (Input:out) to the in port of the Sum plan (Sum:in). The second Link block links the same out port to the in port of the Count plan (Count:in).

Note that we provide default port names in each plan block that combine the plan name and port function, for example “Sum:in” and “Sum:out”. Therefore, students do not need to create port names, and can fill in the port names in the Link blocks by copying from the plan blocks. However, when students use same plan block more than once in their program, they have to change the default port name for different copies of the same plan. For example, if a second copy of the Sum Plan block is used, its default port names must be renamed from “Sum:in” and “Sum:out”. Correct and consistent use of port names is essential for correctness, and the internal implementation of the Link blocks tests for this, and gives a message if the port name filled in the Link block does not match the spelling of its original name in the plan block.

The result of this process (Figure 6) is fully executable, and can be tested and debugged. Note that the execution makes use of buffers: in this example (see the left side of Figure 7), the Input plan runs to completion, collecting all the inputs, then the Sum plan runs to completion (reading from a behind-the-scenes buffer) and computes the Sum of the input, followed by the Count plan counting the number of input values, and so on. This is quite different to how a final (procedural) program executes: a single input is read, and a running sum and count updated before dealing with the next input value. The testing of this step is part of the evaluation and testing schedule for the whole programming process. As students proceed through the process in Figure 3 they maintain a checklist (Table 2). The first three columns are filled according to the results in Table 1. The remaining columns correspond to the steps in the process. After each step, the results from testing are recorded and compared with results from the earlier steps in the process. In this step, the testing and debugging results are filled in the fourth column (Step 2), and Step 2 will be considered to be completed if the test results in the Step 2 column are the same as those in column Step 1.

Test Cases	1, 2, 3, -1	2, 3, 7, 8, -1
Predicted Answers	2	5
Results from analysis of goals and plans (Step 1)	2	5
Results from mapped plan blocks (Step 2)	2	5
Results from expanded plan details (Step 3)		
Results from merged details (Step 4)		
Results from final program (Step 5)		

Table 2: Example of the test schedule

³ Our process also includes guidance for developing new plan blocks, but due to space limitations this is not covered in this paper. We return to this issue in Section 5.

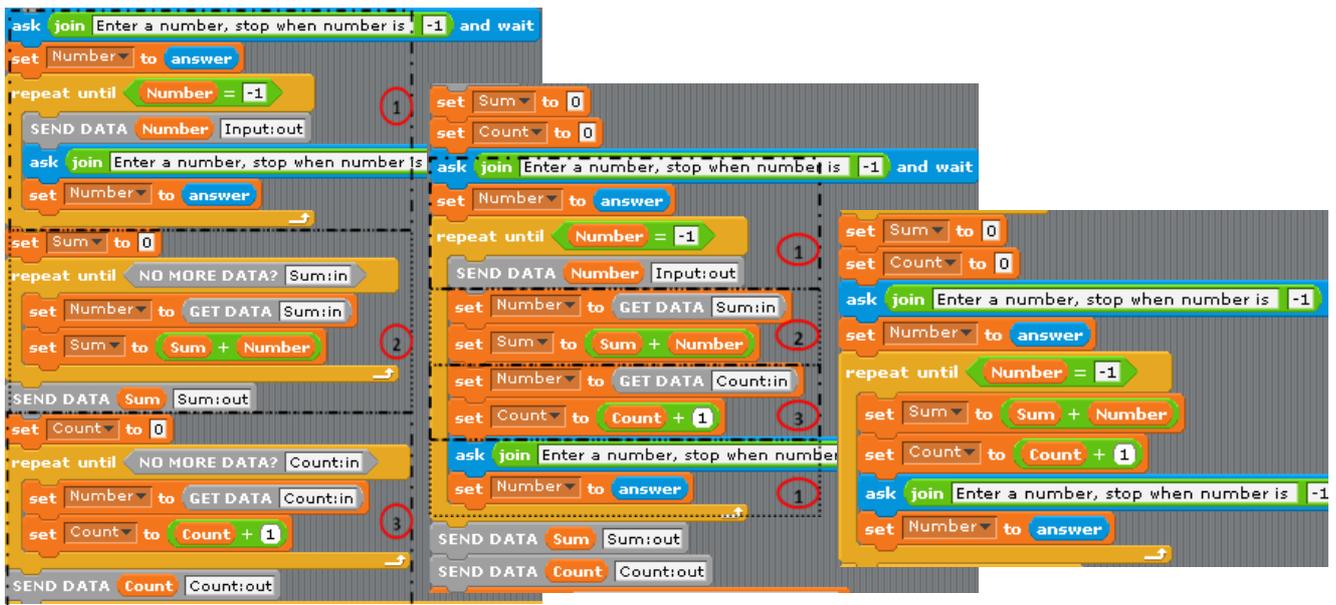


Figure 7: Examples of expanded program (left), merged program (middle), and final program (right)

3.3 Expanding Plan Blocks

Expanding plan blocks means replacing each plan block with the defined details within it (see the left side of Figure 7). Since data flow blocks (“GET DATA”, “SEND DATA”, and “NO MORE DATA?”) inside every plan block contain local parameters “in-port” and “out-port” to refer to ports of their hosting plan block, after expanding from a plan block, these parameters must be changed to the plan port names in order to work with the linkage using plan ports. For example, the blocks details within the Sum plan (which are not shown in Figure 7) include the block “repeat until NO MORE DATA? in-port”, where “in-port” is the first parameter of the plan block. When replacing the Sum plan block with its plan details we replace “in-port” with the value of the plan block’s first parameter, namely “Sum:in” (Figure 6) yielding the statement “repeat until NO MORE DATA? Sum:in” (see the numbers part 2 in the left side of Figure 7).

To help explain this process we have provided students with video clips of a screen capture that demonstrates how to firstly duplicate plan details from each plan block and then how to replace the parameters by copying-and-pasting a port name from the plan block. Whereas the previous two steps require human thought and creativity, this step is purely mechanical and could be automated in future work. Note that the expanded program is also executable and testable.

3.4 Merging Expanded Plan Details

Merging expanded plan details aims to combine the details from different plans into one program in which data flow (and the associated use of buffers) is eliminated, and in its place, a single datum is sent and received between plans. In other words, traditional variables, rather than buffers, are used to communicate data between plans. Common variables are also shared between plans (see the middle part of Figure 7). Since plans all follow the same pattern (iterating, reading items from their input port, and dealing with items one at a

time), they can be merged by following three steps, which are demonstrated to students with video clips of examples. The first step of merging plan details is to collect all the blocks that initialise variables by setting or inputting initial values, and put them immediately after the “End Links” block, i.e. at the start of the merged program. For example, in the middle of Figure 7, the first two statements, “set Sum to 0” and “set Count to 0”, as well as the 3rd and 4th statements, “ask” and “set” to input the initial value for variable Number, are placed immediately after “End Links”.

The second step is to combine loops that share the same data flow. In this situation, the first loop is used to generate the data flow, while the other loops receive this data flow. Hence, the bodies of the other loops can be moved within the first loop. Specifically, consider the case where output port portA is linked to input port portB, and we have the following two loops:

```
Repeat until <condition>
  <body of first loop part 1>
  SEND DATA (value, portA)
  <body of first loop part 2>
End repeat
Repeat until NO MORE DATA(portB)?
  Set Var to GET DATA (portB)
  <body of second loop>
End repeat
```

Then both loops will execute the same number of times because the second loop executes once for each data item sent in the first loop. Therefore, the second loop can be eliminated by moving its body inside the first loop⁴:

```
Repeat until <condition>
```

⁴ This assumes that there are no common variables between the loop bodies, which can be ensured by renaming. Since the loop bodies originate in different scopes, there cannot be common variables. However, there may be variables which use the same name (in different scope), and bringing them into the same scope would require renaming to avoid the distinct variables being conflated.

```

<body of first loop part 1>
SEND DATA (value, portA)
Set Var to GET DATA (portB)
<body of second loop>
<body of first loop part 2>
End repeat
    
```

For example, since the loop of the Input Plan generates a data flow to both the Sum Plan and the Count Plan, three loops (see parts 1, 2 and 3 in the left side of Figure 7) are merged under the loop condition from the Input Plan loop (Repeat Until Number = -1). The loop bodies from both Sum Plan and Count Plan are put after “SEND DATA Number Input:out” and before the blocks for inputting the next value of Number (see the numbered parts (1–2–3–1) of the program in the middle of Figure 7). Note that blocks outside of each loop body, such as “SEND DATA Sum Sum:out” and “SEND DATA Count Count:out”, are still kept outside of the merged loop (see the final two blocks in the middle of Figure 7).

The third step is to remove the loop control where there will only be a single value in a data flow. For example, the loop controls from the Dividing Plan and Output Plan can be removed, since the input data flows to these plans only have single values (as shown in Figure 5, and confirmed in Table 1). Once more, the merged plan details are executable and testable. Table 2 is used to check whether the testing results from Step 4 are the same as those from previous steps.

3.5 Simplifying the Merged Details

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

When a variable has its value sent to an output port, and subsequently another variable receives the same value from a linked input port, the second variable should be consistently renamed to match the first one. When we have code of the form:

```

LINK p1 p2
...
SEND DATA (v1, p1)
v2 := GET DATA (p2)
<code referring to v2>
    
```

Then the variable v2 receives its value (via the SEND and GET) from v1, and can be renamed to v1:

```

LINK p1 p2
...
SEND DATA (v1, p1)
v1 := GET DATA (p2)
<code referring to v1>
    
```

For example, consider “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 renamed to Sum. Similarly, variable Number2 is replaced by variable Count.

This renaming of variables means that the SEND and GET blocks become redundant and can be removed, leaving only variables and control flow blocks, which are independent from scaffolding and data flow blocks. Therefore, the last step is to remove all the scaffolding

blocks, both those used to define links (“Begin Links”, “Link”, and “End Links”), and those used to specify data flow (“NO MORE DATA?”, “GET DATA” and “SEND DATA”). This results in the final program shown on the right of Figure 7. At the end of this process, the final program is tested and the last column of Table 2 shows the test results in Step 5, which should be the same as those in the previous column.

4 Evaluation

We evaluated our approach by comparing the answers to a programming question from exams in an introductory programming course at Tairāwhiti Campus, Eastern Institute of Technology in New Zealand. We collected answers from the final exams in the course from 2006 to 2009, and for 2011 collected answers from the final exam and the mid-term test (Hu, Winikoff, and Cranefield 2012). Note that the programming questions used in the exam were similar across years, for example, calculating the sum and (positive or negative) count, or the average of a sequence numbers, and are thus comparable. Also, note that the programming questions in this course’s exams are done on a computer, rather than on paper.

In all years the course was taught by the first author of this paper. In the institutes of technology and polytechnics in NZ, the introductory programming course is delivered as a total of nine week module for the first year diploma programme. Each week had a three hour mixture of teaching and exercises in a computer lab. The course outlines are listed in Table 3. From 2000–2009 the course taught programming using Visual Basic (VB) (and a conventional approach). In 2011 the course used BYOB. The 2011 course retained a conventional approach for the first half of the course, but adopted our proposed process and tool for the second half (see Table 3).

All the answers on the programming question were remarked using the same criteria: identifying variables, using fragments of key code, combining fragments, and being bug free. The summaries of exam results are shown in Table 4. In order to establish a causation (i.e. students did better *because* of the new method) we consider a range of possible alternative explanations for the performance improvement, and rule them out. We compare data from students using the new method with data from students using the old method, and given an observed difference, we rule out other possible causes, such as different student cohorts from year to year, different exam questions, or changes of computer languages (VB to BYOB). We do this by considering a number of hypotheses.

Our first hypothesis is that the scores of the programming question in final exams for the conventional approach (2006–2009) do not show significant differences (more precisely: come from populations with the same probability distribution). This is the case ($p = 0.689 > 0.05$, see Table 5) and so we conclude that **changes in cohort** from year to year, **and in exam questions** from year to year do not make a significant difference.

Our second hypothesis is that the scores for the conventional approach (including both 2006–2009, and

also the 2011 mid-term exam⁵) do not show significant difference. Again, this is the case ($p = 0.603 > 0.05$), which suggests that the 2011 cohort is not significantly different to earlier cohorts, and also that the use of **BYOB** rather than VB is not in itself the cause of a significant change in performance in the exam.

Wk	2006 – 2009		2011	
	Topics	Topics	Contents	Contents
1	Introduction to Visual Basic	Introduction to BYOB	Pseudocode Computer Languages Sequence: Input, Process, Output Flowchart	Pseudocode Computer Languages Sequence: Input, Process, Output Flowchart
2	Input, Process and Output	Making Decision	Flowchart Desk Check Pseudocode BYOB	Flowchart Desk Check Pseudocode Visual Basic
3	Making Decision	More about Making Decision	Nesting of Selection Flowchart Desk Check BYOB Pseudocode Documentation	Flowchart Desk Check Pseudocode Visual Basic Documentation
4	More about Making Decision	Repeating Actions	Flowchart Desk Check BYOB Pseudocode Documentation Nesting of Repetition	Review Nesting of Selection Solving problems Exercises
5	Repeating Actions	Analysis of Problems	Analysis of Goals Design by Plans	Flowchart Desk Check Pseudocode Visual Basic Solving problems Documentation
6	Integration of Selection and Repetition	Steps of Solving Problems	Solving Problems by Provided Plans	Nesting of Repetition Solving problems Exercises
7	More about Integration	Put All Together to Solve Problems	Build Your Own Plans Solving Problems from Goal to Program	Review Solving problems Exercises
8	Revision	Revision	Revision	Revision
9	Test	Test	Written and practical Test	Written and practical Test

Table 3: Course outline by years

Our last hypothesis is that the new method does make a difference, i.e. that including students' performance in the final exam in 2011 (i.e. after being taught our new approach) *will* result in a significant difference. This is the case ($p = 0.031 < 0.05$), and since we have excluded a change in cohort, or programming language, or exam question, we conclude that our method has made a significant difference.

⁵ Ideally we would compare mid-term tests from all years, but 2006-2009 didn't have a mid-term test.

Year of Exam	Conventional Method					Experiment Method
	2006	2007	2008	2009	Mid-term 2011	2011
Student Numbers	13	16	13	8	7	8
Mean	33.3	53.8	36.8	39.4	52.4	84.5
Median	18	82.5	0	22.5	40	100

Table 4: Summary of student results

Student Groups	p-value
Final exam from 2006 to 2009	0.689 (> 0.05)
Final exam from 2006 to 2009 and mid-term exam 2011	0.603 (> 0.05)
Final exam from 2006 to 2009 and final exam 2011	0.031 (< 0.05)

Table 5: Kruskal-Wallis H test⁶ results

Having determined that including the 2011 final exam leads to a significant difference (i.e. rejecting the null hypothesis), we would like to find out which medians of examination scores are different. We performed a family of pairwise comparisons using the Mann-Whitney U test and Holm's sequential Bonferroni adjustment to reduce the chance of any type 1 errors. We only consider the four comparisons between the samples from 2011 and each of the earlier years (see Table 6). This is because 2011 is the year in which the intervention we wish to measure was applied. We made three hypotheses to the sample data summarized as follows. The Mann-Whitney U Test result of each paired comparison to 2011 is smaller than its threshold p-value, which indicates significant differences of examination scores between the year 2011 and each individual year from 2006 to 2009. Therefore, the evaluation has shown a statistically significant improvement in student performance using our new approach. The difference is not due to variation in the cohort, in the examination questions, or in the use of BYOB.

Paired Comparisons	p-value (Holm-Bonferroni threshold)
Final exam between 2011 to 2006	0.003 (< 0.013)
Final exam between 2011 to 2008	0.01 (< 0.017)
Final exam between 2011 to 2007	0.021 (< 0.025)
Final exam between 2011 to 2009	0.025 (< 0.05)

Table 6: Comparing Paired Examination Scores

There were two limitations in the evaluation. However, even though the number of students was low, there was still a clear and statistically significant result. There was also a ceiling effect (where for the 2011 cohort

⁶ Since we had a small group of students in each year (between 8 to 16) and the performance of novice programmers is known to not follow a normal distribution, we used a non-parametric statistical analysis of variance technique, which does not make any assumptions about the shape of the underlying probability distribution. The Kruskal-Wallis one-way analysis of variance by ranks (H test) is a statistical test for measuring the likelihood that a set of samples all come from populations with the same probability distribution.

five out of eight students produced programs that were awarded full marks). However, this ceiling effect actually *reduces* the difference between the experimental and conventional groups, and there would be a *more* significant improvement if we used an instrument that did not have a ceiling effect. Overall, the evaluation results are significant, but further evaluation would help to strengthen the results.

5 Conclusion

We have introduced a well-defined, iterative and incremental program development process for teaching novice programming. The process provides a guideline for novices to develop from the concepts of goals and plans to final code in a visual environment. The process includes five major steps (Figure 3), which guide the student through a process of stepwise refinement. Each step has strategies, and heuristics to guide novices. A significant difference with existing process approaches is that our process includes feedback from every step rather than having one round of feedback from the final program. This regular feedback from every intermediate step encourages students to continue to progress “from victory to victory” in the next step. Our research suggests that the experimental teaching method proposed, with a well defined process, use of goals and plans, and a visual notation, has the potential to significantly improve learning of programming skills.

Note that we are aiming to teach generic programming in a way that leads to further computer courses. This is why we focus on problems that are more representative of the sort of algorithmic programming done in later courses, rather than the sort of applications that BYOB is typically used for. In other words, BYOB is merely a vehicle, and the process is applicable to other programming languages. One area for future work is to assess whether the cohort that did the course in 2011 (with the new teaching method) did better, worse, or the same in subsequent programming courses. We have done a preliminary comparison of the overall exam mark in the second programming course (PP590) for the 2006-2011 cohorts. However, although the PP590 exam marks for the 2011 (experimental) cohort are higher (average of 53.857 for 2011, compared with 34.1 across 2006-2009 [2007 is lowest with 23.67, 2008 highest with 44.308]), the difference is not statistically significant. Note that the PP590 exam typically includes both questions that involve programming, and questions that assess knowledge rather than programming skill (e.g. “what is pseudocode?”). This means that the overall exam mark is not a good measure of programming ability. Unfortunately, we do not have the marks for individual questions, only the total exam mark.

As noted earlier, where the library is missing a required plan block, the student must develop it themselves by modifying similar blocks (e.g. multiplication to division), or by using a template-based process (not described in this paper). Our experience has been that this is not an issue (as indicated in the evaluation results), and we argue that constructing a single plan block can be expected to be easier than constructing an entire program, i.e. that even where some plan blocks are not in the library, our process has the

effect of reducing the problem to a smaller one. However, more broadly, this is a limitation of our work: we assume that the task at hand is reasonably well covered by the library of plans. Another limitation of this work is that the model of a plan network with data flowing between them is not expected to be applicable to all programming tasks. However, since our aim is to help novices to learn basic programming skills, we do not see the lack of universal applicability as a significant issue.

Having a programming process is, to some extent, a trade off in that the process is more structured (and hence more easily followed by novices), but also more complex than unguided programming. Our experience, and results, clearly show that the process is usable, and furthermore, that the benefits from having a structured process outweigh the costs of the additional complexity. We argue, as Kölling and Henriksen (2005) did, that without a programming process we could end up with two groups of students: those who fail programming, and those who pass by their own implicit process. We draw an analogy with swimming lessons. We would not let swimmers just jump into a river or the sea to learn swim. Instead, we prefer to teach them steps of swimming in a well-designed style at a swimming pool in the first place.

During the teaching, we recognised that some students were reluctant to follow the process they were taught (using goals and plans), and instead used BYOB blocks directly. It is not clear whether these students did not need the detailed process: they might be in the group who can find their own process implicitly (Kölling and Henriksen 2005). However, what is clear is that across the class, the new process *did* make a difference. It is possible that for some of the students the process assisted them to advance to a point where they no longer needed to follow the explicit process for simple programs. We also note that it may be easy to build up a program for a simple problem without explicit process, but it is hard to directly write a program for a complex problem.

There are a number of directions for future work. One direction concerns “bricoleurs” (Turkle and Papert 1990) who prefer to arrange and rearrange existing material transparently. To what extent does our process support or hinder this style of work? We argue that the steps of expanding and merging can actually expose and deal with the details of goals and plans, and that the feedback from each step should support a bricoleur style of negotiating and renegotiating. However, more work is needed to confirm this.

Another direction for further work concerns the number of plans. A limitation of our plan framework is that we only provided a limited number of plans. In other words, there are not many different plans to choose for the same goal. Therefore, one area of further investigation is to investigate students’ mental models in order to better understand how they select plans when many plans are available.

Finally, one weakness with the process is that the plan merging process, although well-defined, is somewhat complex. Therefore an area for future work is to investigate how to better support the plan merging process. However, we want to provide support that helps students to gain insight into the merging process, rather than just providing a “wizard” that does the merging of

plans. On the other hand, the expansion of plans (Section 3.3) can, and should, be automated.

6 References

- Ben-Ari, M. (2001): Program visualisation in theory and practice. *UPGRADE*, 2, 2, 8-11.
- Caspersen, M. and Kölling, M. (2009): STREAM: A First Programming Process, *Transactions on Computing Education (TOCE)* 9, 1, 4:1-29, ACM
- de Raadt, M. (2008): *Teaching programming strategies explicitly to novice programmers*. PhD Thesis, School of Information Systems, University of Southern Queensland.
- de Raadt, M., Watson, R. and Toleman, M. (2006): Chick Sexing and Novice Programmers: Explicit Instruction of Problem Solving Strategies. In *Proceedings of the Eighth Australasian Computing Education Conference (ACE2006)*, 52, 55-62.
- Ebbrahimi, A. (1992): VPCL: A Visual Language for Teaching and Learning Programming. (A Picture is Worth a Thousand Words). *Journal of Visual Languages and Computing* 3 (1992), 299-317
- 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.
- Guzdial, M. & Soloway, E. (2002): Teaching the Nintendo generation to program, *Communications of ACM*, 45, 4, 17-21
- Hu, M., Winikoff, M. and Cranefield, S. (2012): Teaching Novice Programming Using Goals and Plans in a Visual Notation, In *Proceedings of the Fourteenth Australasian Computing Education Conference (ACE2012)*, 123, 43-52.
- Janzen, D.S. and Saiedian, H. (2006): Test-driven learning: Intrinsic integration of testing into the CS/SE curriculum. In *Proceedings of the 37th Technical Symposium on Computer Science Education (SIGCSE'06)*, 254-258. ACM.
- 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.
- Kölling, M. and Henriksen, P. (2005): Game Programming in Introductory Courses with Direct State Manipulation, In *Proceedings of the Innovation and Technology in Computer Science Education (ITiCSE'05)*, 59-63. ACM.
- 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*, 2, 2 (June 2011), 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.
- Mayer, R. (1981): The psychology of how novices learn computer programming, *Computing Surveys*, 13, 1, 121-141, ACM
- Naps, T. L., Rößling, G., Almstrum, W., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S., and Velazquez-Iturbide, J. A. (2003): Exploring the Role of Visualization and Engagement in Computer Science Education. *ACM SIGCSE Bulletin*, 35, 2, 131-152.
- Pattis, R. (1990): A philosophy and example of CS-1 programming projects, In *Proceedings of the Twenty-first SIGCSE*, 34-29, ACM
- 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. In *Proceedings of the working group reports on Innovation and Technology in Computer Science Education (ITiCSE'07)*. ACM.
- Porter, R. and Calder, P. (2003): A pattern-based problem-solving process for novice programmers. In *Proceedings of the Fifth Australasian Computing Education Conference (ACE2003)*, 20, 231-238.
- 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, 163-207.
- Turkle, S and Papert, S. (1990): Epistemological Pluralism: Styles and Voices within the Computer Culture, *Journal of Women in Culture and Society*, 1, 16, 11, 128-157.
- Utting, I., Cooper, S., Kölling, M., Maloney, J., and Resnick, M. (2010): Alice, Greenfoot and Scratch – A discussion. *ACM Transactions on Computing Education*. 10, 4, Article 17, 11 pages.
- Winslow, L. E. (1996): Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin*, 28, 3, 17-22.

It's Never Too Early: Pair Programming in CS1

Krissi Wood
School of ICT
Otago Polytechnic
Dunedin, New Zealand
Krissi.Wood@op.ac.nz

Dale Parsons
School of ICT
Otago Polytechnic
Dunedin, New Zealand
Dale.Parsons@op.ac.nz

Joy Gasson
School of ICT
Otago Polytechnic
Dunedin, New Zealand
Joy.Gasson@op.ac.nz

Patricia Haden
School of ICT
Otago Polytechnic
Dunedin, New Zealand
Patricia.Haden@op.ac.nz

Abstract

This paper describes the use of the Pair Programming software development methodology in the earliest weeks of a first programming course. Based on a broad, subjective assessment of “programming confidence”, instructors placed students in level-matched pairs for a portion of their programming exercises. Students who began at the lowest levels of confidence showed significantly better exercise completion rates when paired than when working individually. Student response to the Pair Programming technique was uniformly positive, and teaching staff report pedagogical, mechanical and social benefits from the practice. These data indicate that successful programming pairs can be constructed based on tutors’ subjective judgements of student performance very early in CS1, before exam scores or code quality assessments are available. Thus Pair Programming can be an effective classroom intervention even with extreme novices.

Keywords: Programming education, Pair Programming, Novice programmer.

1 Introduction

Failure rates in first computer programming papers (usually called CS1) are alarmingly high, often greater than 40% (Bennedsen and Caspersen, 2007). Recent work (Robins, 2010) has identified student struggles in the first days and weeks of CS1 as a significant contributing factor to this high failure rate. Robins has demonstrated mathematically that students who fail to acquire the core concepts presented in first programming lessons are frequently unable to recover, leading to high drop out and failure rates. He maintains that this is largely due to the scaffolded structure of computer programming, where each skill builds upon, and requires mastery of, a set of simpler skills. Thus it is essential that we find classroom approaches and interventions that can support novice programmers during their earliest teaching sessions. In the current study, we explore the possibility of leveraging a specific programming methodology – Pair Programming – in the very first weeks of CS1. To do this,

we introduce a protocol for assigning students to pairs using holistic judgements made by in-class teaching staff. These judgements were made after the second week of CS1 before either exam marks or code quality assessments were available. As detailed below, this pairing protocol resulted in significantly better class performance for those students who initially appeared to be at greatest risk.

Pair Programming is a formal software development protocol where two programmers work synchronously on a single piece of code (Williams and Kessler, 1998). The protocol includes detailed policies for participant roles and procedures. One member of the pair is the Driver, who controls the mouse and keyboard, physically creating the code. The other member of the pair is the Navigator, who oversees the construction process, watches for errors, makes suggestions and locates resources. Partners switch roles at regular intervals, usually every 15 to 20 minutes. Pair Programming originated in industry but has, in the last decade, become increasingly common in the classroom. An active research community is exploring the potential benefits of Pair Programming to students and teachers, while considering mechanical and procedural issues in its use.

Studies have shown that Pair Programming can contribute to an improvement in learning outcomes. In a large longitudinal study involving several thousand students, McDowell, Werner, Bullock, and Fernald (2004) found that students in classes that used Pair Programming were more likely to complete their classes and to continue in a computer science major than were students in comparable classes that used only solo programming. Students from the Pair Programming classes had equivalent exam performance to solo students, addressing the concern of some educators that Pair Programming permits one student to “freeload” on a stronger partner.

Similarly, Mendes, Al-Fakhri and Luxton-Reilly (2005 and 2006) have performed two large-scale studies of Pair Programming at the University of Auckland. In these studies, students in Pair Programming classes performed better on programming exercises, and earned higher exam marks, than solo programming controls.

Williams (2007) describes the lessons learned in seven years of using Pair Programming at a large university in a variety of Computer Science papers at all academic levels, including graduate. Williams details benefits of the protocol for both teachers and students. For students,

Copyright © 2013, Australian Computer Society, Inc. This paper appeared at the 15th Australasian Computing Education Conference (ACE 2013), Adelaide, South Australia, January-February 2013. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 136. A. Carbone and J. Whalley, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Pair Programming supports the building of stronger social relationships (through the need to work together), increases retention, and reduces “waiting time” for teacher feedback as two students working together can often resolve a problem for which a student working alone would require teacher assistance. For teachers, the protocol reduces marking time (by halving the number of submitted assignments), reduces student demand in practical sessions, and improves general work ethic by, they hypothesise, engendering a sense of mutual responsibility between partners.

Brought, Wahls and Eby (2008) performed a tightly-controlled study of Pair Programming. In a large programming paper with multiple sections, they randomly assigned some sections to use Pair Programming and some to use solo programming. Students enrolled in a section without knowing which method would be used in that class, and were not allowed to transfer between sections after the start of the paper. Braught et al. compared code quality on individual assignments, as well as exam marks and time to complete assignments. They found an interaction between programming protocol and scores on the Scholastic Aptitude Test (a test of general academic level administered prior to college or university entrance in the United States) such that greatest benefit of Pair Programming was seen for students with lower SAT scores. This implies that Pair Programming might be especially helpful for those students who would otherwise struggle with a programming paper, which is in accordance with the higher retention and completion rates seen in many Pair Programming studies.

In all of the preceding studies, (and in others discussed below) subjective student feedback was gathered, asking students for their views of the Pair Programming experience. Student feedback is nearly universally positive, with students reporting that they enjoy working in pairs, that they feel they can program more quickly with a partner, that they are less likely to “get stuck”, and that they appreciate the opportunity to get to know fellow students through working together. Negative feedback (and less positive performance outcomes) occurs primarily in the case of dysfunctional pairings, that is, when partners are unable to work effectively together.

Although some of the reported benefits of Pair Programming can be obtained simply through random pairing (e.g. McDowell et al., 2004 used only random pairing) there is compelling evidence that careful selection of pairs reduces the probability of dysfunctional pairings. Specifically, both educational benefit and student satisfaction appear to be maximised when the two members of a pair have similar levels of programming ability.

In the long-term study described by Williams (2007), teachers experimented with a variety of metrics to determine pairings, including standardised general exam scores, grade point average, the results of personality and self-esteem tests, learning style scores, and work ethic (based on self-report). They paired students in various combinations of these measures, using both similarity and dissimilarity of scores. The most successful pairings were those based on similar mid-term exam score, the most direct measure of a student’s programming skill at the

time of the pairing. On self-report, Williams’ students consistently request to work with a student of equal or greater programming skill. Since it is not possible to give one member a stronger partner without giving the other a weaker partner, Williams recommends attempting to pair students of equal skill levels.

Cliburn (2003) explored directly the effect of partner similarity by constructing highly dissimilar pairs. He originally paired students “from different cultural or ethnic backgrounds [and]...upper with lower classmen”. The result was poor collaboration and poor exam performance. He then re-paired students based on their project marks, matching students with similar results. With these pairings he observed better project quality and completion rates, and higher exam scores.

Direct inspection of students’ experience of Pair Programming also shows the advantage of pairing students of similar ability. Chaparro, Yuksel, Romero and Bryant (2005) used a variety of metrics to explore students’ qualitative views of Pair Programming. Through the use of observation, questionnaires, semi-structured interviews and field notes they determined that students prefer, and find most effective, pairings of similar skill levels. Katira, Williams and Osborne (2005) queried students directly about the “compatibility” of their Pair Programming partner. Students rated as more compatible those partners whom they perceived to be of similar skill. Students’ perception of the skill levels of their partners was accurate, as measured by exam scores and grade point average.

More recent studies (e.g. Radermacher and Walia, 2011 and 2012; Braught, Wahls and Eby, 2008) have accepted pairing by skill level as the appropriate default, citing the accumulating evidence in its favour.

While there is a growing consensus that pairing by skill level produces the most successful Pair Programming experience, the measurement of skill remains problematic. As we are interested in the use of Pair Programming very early in a first programming course – ideally in the first weeks – we require a measure of ability to be made before exam or major project scores are available. We have thus used a subjective metric, based on instructor observation of student performance, which can be made in the first weeks of the semester.

Our observational assessment of ability is based on what we call “programming confidence”. The term “confidence” in this context is not a personality metric; it does not, in our experience, correlate with self-esteem. It is a description of the way in which students approach programming exercises. The confident student programmer approaches coding exercises boldly, is willing to experiment with the techniques being learned, is relatively unfazed by coding errors and seems to expect to be able to solve the assigned problem. These students may have prior programming experience in school or as a hobby, or they may have a history of success in contexts they perceive to be similar to programming (e.g. games or puzzles), or they may simply feel comfortable with the particular intellectual exercise involved. Student programmers who lack confidence are less able to make independent progress with coding exercises. They frequently become “stuck”, and will wait for assistance from the instructor, rather than try an alternative approach

on their own. This slows their work pace, and often makes it difficult for them to complete in-class assignments in the allotted time. Programming confidence, as we define it here, reflects current programming ability, and can change rapidly as the student gains experience. We have observed that students who start out with little confidence can eventually develop considerable programming skill, if they are able to navigate successfully the difficult early stages of learning. In section 2, we discuss further the process used to make our assessments of student programming confidence.

It is interesting to note that Thomas, Ratcliffe and Robertson (2003) attempted to place students on an equivalent continuum of programming confidence by self-report. Each student was asked to rate himself or herself on a 10-point scale from “Code Warrior” to “Code-Phobe”. Thomas et al.’s description of these terms is extremely close to our conceptualisation of programming confidence. Based on the students’ own rating, Thomas et al. compared the efficacy of same vs. opposite pairings. That is, in one condition they paired two high scoring students or two low scoring students; in the other condition, they paired a high-scoring student with a low-scoring student (middles were always paired with other middles). They report that Same pairings perform better than Opposite pairings on coding exercises, and that students consistently prefer being paired with someone at their own level on the Warrior-Phobe scale.

Another factor that has been explored as a potential determinant of the efficacy of Pair Programming is the time course of the pairing. McDowell et al. (2004) paired students for an entire semester, and pairing was used on both in-class and out-of-class assignments. Radermacher and Walia (2011), in contrast, paired students only for a single 50-minute class session. Based on their lengthy experience with Pair Programming, Williams (2007) and her colleagues (see for example, Nachiappan, Williams, Ferzli, Wiebe, Yang, Miller and Balik, 2003) recommend switching pairs often. They note that this reduces the impact of any dysfunctional pairing and increases the social benefit which many students cite as an advantage of the method. They further advise that Pair Programming be initially used only in-class, until students have mastered the technique. This has the added benefit of eliminating scheduling difficulties, which are noted as problematic by many students in studies using out-of-class exercises (cf. McDowell et al., 2004; Hanks, 2006).

Thus, following current best practice for the implementation of Pair Programming in the classroom, we intend to pair students based on programming confidence (as defined above), to include a combination of paired and individual exercises during the semester, and to change pairs for each Pair Programming session. In this way we hope to be able to use Pair Programming in the very earliest stages of programming education, where it is hypothesised that students are at greatest risk of failure (cf. Robins, 2010).

2 Method

The study was conducted during a one semester (16 teaching weeks) offering of a first programming course at

Otago Polytechnic in New Zealand. “Programming 1” is a required paper in the first semester of our Bachelor of Information Technology degree. For the majority of students it is their first exposure to formal computer programming, although there are generally a small number of students who have previously taken a programming paper (some who have previously taken Programming 1 but not passed), and occasionally students with hobbyist coding experience. In this offering, 40 students started the paper, including 3 repeaters and 11 with some other prior programming experience.

The focus of Programming 1 is on programming fundamentals, such as variable manipulation and flow of control. The paper is taught in C# using Visual Studio, but is taught exclusively on the console, and contains only minimal Object-Oriented theory (formal OO and GUI work begins in our Programming 2 paper in second semester). Programming 1 comprises two two-hour sessions each week. In a typical session, a new topic is introduced by the lecturer with discussion and code examples. Students are then given a set of practical exercises to perform in class on the discussed topic. Practicals are designed to be completed during class by the majority of students.

In previous offerings of Programming 1, each student worked individually on all practical sessions. In the semester in which this study was conducted, Pair Programming was introduced in selected sessions. Our goal was to begin Pair Programming as early as possible, matching students at comparable levels of ability, as dictated by the current literature. While the common quantitative metrics of ability – exam scores and code quality assessment – are not available in the first weeks of CS1, our experience as programming instructors convinced us that there were observable differences between students even in these early stages. These differences we have summarised as “programming confidence” (see discussion above). We hypothesised that programming confidence could be a criterion for the construction of successful pairs. Further, we believed that judgements of programming confidence could be made simply through observation of student behaviour by experienced programming educators. This hypothesis was based on our conviction, developed over some 40 years of combined CS1 teaching experience, that “we know it when we see it”. Thus we determined to assign students subjectively to one of three levels of programming confidence, and to use this assignment to construct programming pairs.

Teaching staff predicted that they needed at least four teaching sessions to identify accurately each student’s level of programming confidence. Thus, for the first four sessions (i.e. the first two weeks of the semester), students worked individually while teaching staff carefully observed their behaviour.

The four session topics were:

1. Introduction to the IDE and writing to the screen.
2. Introduction to variables and reading from user input.

3. Introduction to data types and computation.
4. Small interactive program combining reading user input, performing computation using multiple data types, and writing output.

After the fourth session, the two classroom tutors made their initial confidence assignments individually. Each student was assigned to a confidence band, with 1 being lowest confidence, 3 being highest confidence and 2 being intermediate. These assignments were made subjectively, reflecting the tutor's sense of how confidently each student approached the programming exercises. Where there were disagreements between the two tutor judgements, the final banding was made collaboratively through detailed discussion of each student's progress and consideration of the number of in-class exercises the student had been able to complete. Prior to the banding, tutors had anticipated difficulty assigning students who fell at the borders of the banding categories. In practice, while tutors had some uncertainty at the boundary between levels 2 and 3, they had no difficulty identifying those at level 1 and there were no disagreements between the two teaching staff about who belonged in this category.

Although no specific quantitative metrics were used to determine confidence bandings, the in-class tutors identified a number of behaviours which they both used consistently to identify low confidence students. These included:

- Getting stuck: The student simply stops working and either switches to some non-related task or waits passively for tutor assistance.
- Copy-coding: The student begins reproducing code samples verbatim where they are not appropriate.
- Frantic random changes: The student begins inserting and deleting code elements randomly in the hopes that an error will be resolved, without any organised plan.

The consistency of assignment to level 1 by both tutors even in the absence of specific quantitative metrics is notable. The very low confidence student seems almost qualitatively different from his peers, at least in the perception of an experienced programming teacher. In future semesters, we intend to analyse formally the initial judgements of the two classroom staff to obtain a statistical measure of inter-rater reliability.

For the next four weeks of the semester, the two classroom sessions each week were handled differently. In the first session students worked individually; in the second session, students were assigned to pairs and used a formal Pair Programming code development methodology. (The technique was explained prior to the first Paired session.) Students were paired based on banding such that each student worked with a student at the same confidence level. Each student was assigned a different partner for each of the four paired sessions. The pairing assignment was made by the instructors prior to

the class session and announced at the beginning of practical work time. In cases where an odd number of students necessitated a cross-banding pairing, this was arranged by the instructors based on their assessment of the students' suitability. Where an odd number of students required one student to work alone, this role was always given to a more experienced Level 3 student. For each session, instructors recorded exercise completions and observed student behaviour.

At Week 6, after four weeks of using Pair Programming in alternating sessions, student feedback was collected. See below for details. Additionally, students were rebanded at this time. The course instructors had noted that different students were progressing at different rates (as is generally true in Programming 1) and some students who had initially been placed in the same band were now working at different levels of confidence. The rebanding used, as much as possible, the same criteria as the original banding. That is, students who were still obviously struggling were assigned to Level 1, and those who were working independently were assigned to Level 3. The new banding was not based on a student's ranking relative to the rest of the class. Thus it was technically possible that the second banding would have no Level 1 students. In actual fact, the second banding produced 8 Level 1 students (22%), 22 Level 2 students (61%) and 6 Level 3 students (17%). See below for a more detailed discussion of the changes in banding over time.

Weeks 7 to 9 of the paper were spent in revision and preparation for the mid-term exam, so no formal practical sessions were held. After the mid-term exam, students were banded based on their exam score to provide an external comparison for the instructors' subjective bandings. Students scoring 55% or lower were considered Level 1, students scoring 55% to 75% were considered Level 2, and students scoring more than 75% were considered Level 3.

The purpose of the second banding (and the banding based on exam score) was to prepare for pair assignments in the remaining weeks of the semester, where we intended to continue the alternation of individual and paired practical sessions. However, students began to express a preference for working in pairs rather than individually. In view of this attitude, and given the positive impact of Pair Programming that was observed during the first experimental weeks (see below) the instructors decided that educational efficacy took precedence over data collection, and did not require students to perform any practicals individually after week 11. The instructors continued to place students into pairs for the planned Paired sessions if they had not self-paired, but students were also allowed to construct their own pairs. Thus only weeks 3 to 6 (inclusive), 10 and 11 are included in the analysis.

3 Results¹

3.1 Practical Lab Completions

During the six experimental weeks, there were six individual and six paired practicals. The mean number of individual practicals completed on time per student was 4.58; the mean number of paired practicals completed on time was 4.97 ($F_{1,34} = 2.489$; $p < .05$).

The distribution of the difference between numbers of paired and individual lab completions across students is shown in Figure 1. Of the 14 students who completed equal numbers of individual and paired practicals, 8 (57%) completed all twelve labs. This apparent ceiling effect compromises our ability to sensitively observe the impact of Pair Programming for students at the top end.

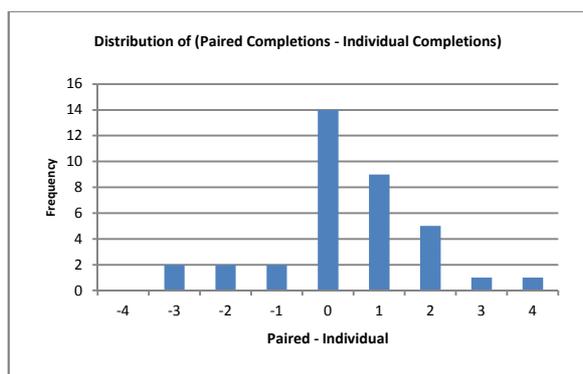


Figure 1: Distribution of Completions

To observe more closely the differential impact of pairing on students of different initial confidence, we can compare completion rates for students based on their first bandings. Due to the low number of students initially banded at Level 3 who did not withdraw from the paper prior to the midterm exam, we combine Levels 2 and 3 for this analysis. Students who had initially been banded at Level 1 completed on average .84 more paired labs than individual; students initially banded in Levels 2 or 3 completed on average .12 fewer paired labs than individual ($F_{1,34} = 4.11$; $p = .05$). This pattern does not seem to be attributable entirely to a ceiling effect, as the total mean labs (out of 12) completed for initial Level 1 students is 9.05, and for initial Level 2/3 students is 10.11. This difference is not significant ($F_{1,34} = 2.53$; $p = .12$). Thus the benefit of Pair Programming as measured by practical lab completion rates appears to be primarily for those students who initially exhibited the greatest difficulty with programming.

3.2 Programming Confidence Bandings

The proportion of students at each Level for each of the bandings is shown in Figure 2. The proportion of students at Level 1 decreased between week 2 and 6, while the proportion at Level 2 increased. Assuming that programming confidence increases with experience, this pattern is as expected. The mid-term banding shows a steep increase in the proportion of students placed at Level 3. Since this banding was based not on instructor

judgment (as the Week 2 and Week 6 bandings were) but on exam score, it is not possible to determine whether this shows an actual continuation of the trend of increasing confidence, or is just a reflection of a comparatively easy exam.

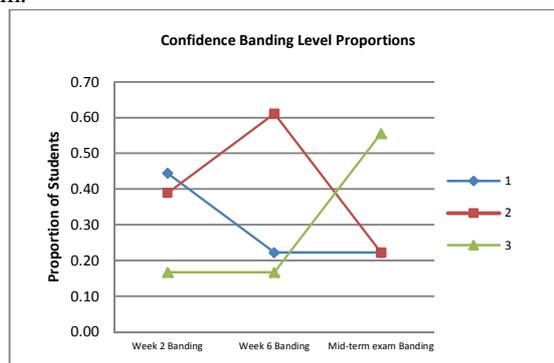


Figure 2: Proportion of students at each level for each banding.

As discussed above, students initially placed in confidence Level 1 turned in significantly more practical labs from paired than from individual sessions, while students initially placed in confidence Levels 2 and 3 did not. This indicates that the instructors' subjective ratings of programming confidence do correspond to some student quality relevant to performance in Programming 1. To interpret this pattern fully, it will help to explore precisely what is being measured in the instructors' confidence judgements. The difficulty of accurately predicting, or even measuring, programming skill has been discussed widely (see, for example, McCracken et al., 2001) and complicates all research into programming education. It would be useful to discover that something as simple as tutor observation could be used to make such a prediction.

If early programming confidence is a useful predictor of eventual programming performance, we would expect to see a correlation between initial banding judgement and final course mark. This was not observed (Spearman- $r = -0.17$; ns). However, it is interesting to consider student performance not just as a function of initial confidence, but as a function of the *change* in confidence seen between Week 2 and Week 6. Since confidence banding judgements were absolute, not relative, we would have hoped to see all students' confidence scores improving with experience, and this pattern was seen generally in the summary of proportions shown in Figure 2, where many students moved from Level 1 at Week 2 to Level 2 at Week 6. However, not all students' banding scores did increase. In fact, of the 36 students who earned final marks in the paper, 20 (56%) actually maintained the same confidence banding from Week 2 to Week 6 (33% went up; 11% went down). Perhaps when predicting eventual programming skill it is useful to look not only at where the student starts, but how rapidly he or she gains programming confidence. To assess this, we can look at the relationship between students' change in confidence in the early weeks of the paper, and their eventual final course mark. For this analysis, we omit the 6 students originally at Level 3 since it was not possible for them to increase their confidence band. Of the

¹ To allow comparisons between analyses, four students who withdrew from the paper prior to the midterm exam have been omitted from all results summaries.

remaining students, one student’s banding dropped from Week 2 to Week 6, 14 stayed the same, and 11 improved. The mean final course marks for the three groups are shown in Figure 3. Those who improved their confidence rankings from Week 2 to Week 6 earned significantly higher final course marks, on average ($F_{2,23} = 3.4; p=.05$).

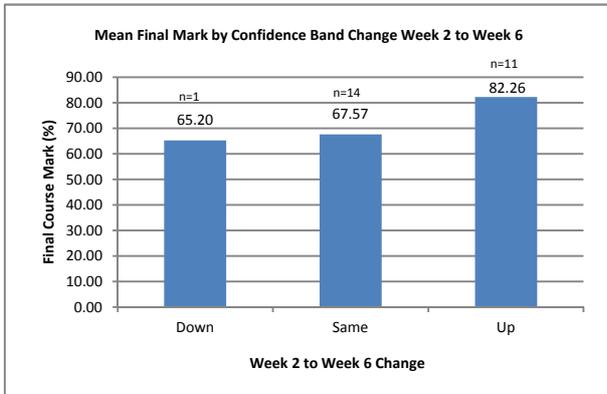


Figure 3: Mean final mark by confidence band change.

3.3 Student Feedback

After the first four weeks of alternating individual and Pair Programming sessions (at Week 6), students completed a brief questionnaire covering their attitudes toward the Pair Programming techniques. The questionnaires were submitted anonymously, and were administered by a non-teaching member of the research team. The questions asked are shown in Table 1.

1.	Which do you enjoy more: pair programming or working alone? Why?
2.	Do you feel you program better in a pair or on your own? Why?
3.	What did you <i>like</i> about the pair programming sessions?
4.	What did you <i>dislike</i> about the pair programming sessions?
5.	Would you like to continue to use pair programming during the remainder of the semester?
6.	Which best describes your programming education experience prior to this paper? 1) No prior experience 2) Hobbyist or self-taught 3) Have taken one or more previous programming papers.
7.	Any other comments?

Table 1: Feedback questionnaire Week 6

After Week 12 of the paper, feedback was again collected. Since prior experience was not expected to be as relevant, given that even complete novices had been through 12 weeks of programming education, Question 6 was replaced with a question designed to elicit students’ opinions about how best to construct a pair: *“Think about the most effective pairings that you have been in this term. What do you think makes a Pair Programming partnership successful?”*

Figures 4 to 6 show summaries of responses to the three binary questions (numbers 1, 2, and 5 in Table 1) comparing Week 6 and Week 12.

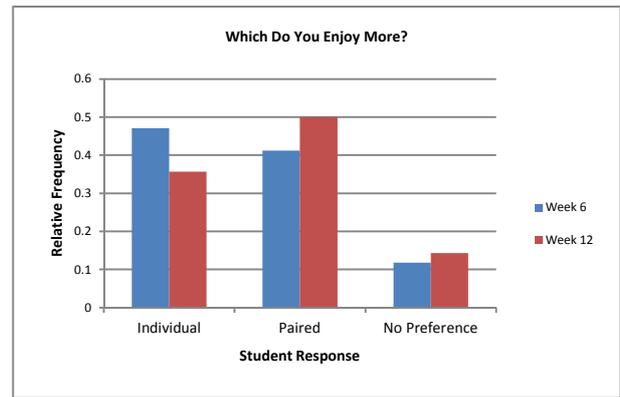


Figure 4: Student preference

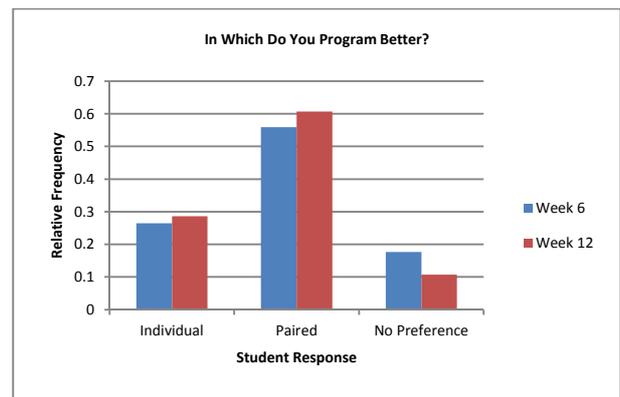


Figure 5: Student judgement of quality

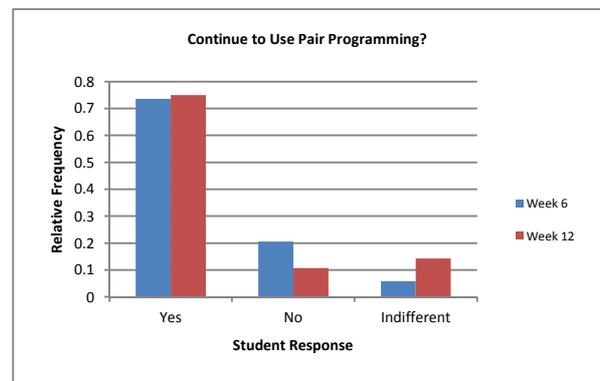


Figure 6: Student willingness to continue

There are no significant differences between the patterns of responses to these questions at Weeks 6 and 12 (by χ^2). The main effect of response collapsed across Weeks is significant for all questions (by $\chi^2; p<.002$).

In Week 6, a greater proportion of students preferred working individually to working in pairs than at Week 12 (47% to 41% at Week 6; 35% to 50% at Week 12). Although this effect is not statistically significant, the trend corresponds to the classroom instructors’ observation that students became more comfortable with the protocol over time. Based on students’ free comments (see below) this appears to be due both to a reduction in social awkwardness as students get to know each other, and increased value of the protocol as the programming tasks become more challenging.

In both Week 6 and Week 12, a greater proportion of students felt they “programmed better” in a pair (58% to

27% collapsed across weeks). Student free comments identify a number of possible rationales for this, including the sharing of ideas, greater opportunity for code checking and increased motivation to do well. In both weeks the majority of students stated that they wished to continue to use Pair Programming during the remainder of the semester (74% to 16% collapsed across weeks). Student free comments show a number of caveats, however, primarily an unwillingness to work with partners who were perceived as weaker programmers.

The reluctance of students to work with a weaker partner can also be seen by looking at the pattern of responses in Week 6 to the three binary questions as a function of self-reported experience level (Question 6 in the Week 6 survey). Figures 7 to 9 show these results.

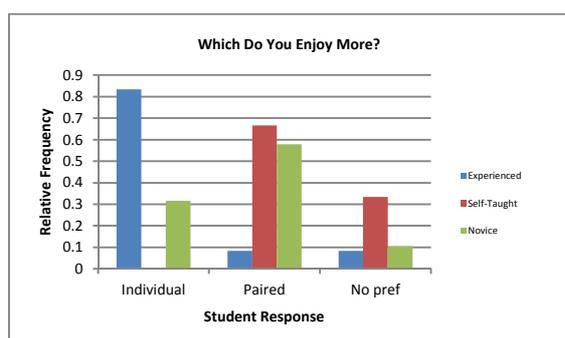


Figure 7: Student preference by previous experience

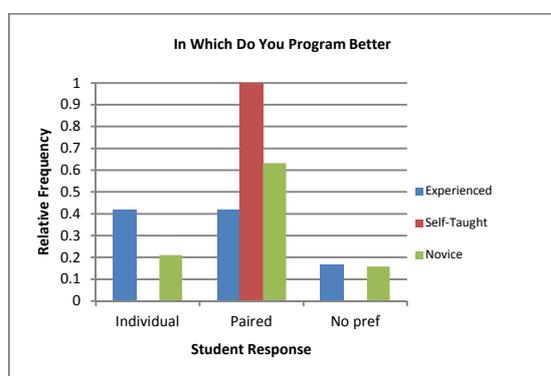


Figure 8: Student judgement of quality by previous experience

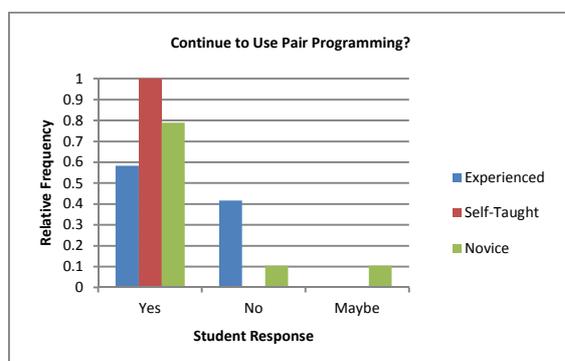


Figure 9: Student willingness to continue by previous experience

Students identified themselves as Experienced ($n=12$), Self-taught ($n=3$) or Novice ($n=19$). Students who classified themselves as Experienced were significantly more likely to prefer working alone than were their less experienced classmates (by χ^2 ; $p<.02$). A similar trend of reluctance of the Experienced students to work in pairs was seen in the questions about programming quality and desire to continue using Pair Programming, but these effects were not statistically significant (by χ^2).

In the remaining survey questions students were asked to identify specific things that they liked and disliked about Pair Programming, and to provide any further comments they wished to make. There was good consistency among student comments, and we were able to identify a small number of comment categories. The complete comment coding for Week 2 and Week 6 is given in Table 2. For each general class of comment, Table 2 shows the proportion of students who made the comment in each week, and the change in proportion from Week 6 to Week 12.

		Week 6 n=33 Pr (Wk 6)	Week 12 n=27 Pr(Wk 12)	Change	
Adv. Individ.	Can work at own pace	0.12	0.11	-0.01	
	More effective learning	0.30	0.11	-0.19	
	Can use own methods	0.09	0.15	0.06	
	Adv. Pair	Allows discussion	0.06	0.04	-0.02
		Builds sense of community	0.39	0.26	-0.13
		Faster	0.42	0.33	-0.09
		More fun	0.03	0.04	0.01
	Disadv. Pair	Can get help when stuck	0.52	0.44	-0.07
		Can learn from explaining	0.06	0.07	0.01
		More code checking	0.15	0.15	0.00
Motivating		0.03	0.04	0.01	
Can get other viewpoints		0.27	0.59	0.32	
Disadv. Pair		Boring for navigator	0.06	0.07	0.01
		Enforced social interaction	0.24	0.15	-0.09
		Evaluation apprehension	0.06	0.15	0.09
		Partners can be incompatible	0.06	0.15	0.09
		Don't like working with stronger partner	0.06	0.04	-0.02
	Classroom is too noisy	0.03	0.00	-0.03	
Don't like working with weaker partner	0.21	0.07	-0.14		

Table 2: Summary of student comments

In Week 6, the most commonly mentioned advantage of Pair Programming was that one could get help from the partner when stuck (mentioned by 52% of respondents). Often the note “instead of having to wait for the lecturer” was added. Novice programmers traditionally need a great deal of assistance, and in large classes it can be difficult for an instructor to respond to all requests in a

short time. While one might expect that two novices working together would not be able to provide useful support to each other, this does not seem to be the case based on this feedback.

In Week 12, the “able to get help” comment was still often made (mentioned by 44% of respondents) but at this point the most frequently cited advantage of Pair Programming was the ability to get another person’s viewpoint and suggestions (this was often phrased as “two heads are better than one”). This shift seems to reflect students’ increasing independence from the lecturer between Weeks 6 and 12.

In both weeks, students often noted that working as a pair was faster (42% and 33% in Weeks 6 and 12 respectively), and that it built a sense of community among the students (39% and 26%) as it required them to meet and get to know their classmates. This impact on the social dynamics of the classroom was among the features that the instructors found most salient during this semester (see further discussion below).

The most frequently cited disadvantage of Pair Programming in Week 6 was the difficulty of working with a weaker partner (mentioned by 21% of respondents). By Week 12, this had fallen to only 7%, perhaps indicating that some of the novice programmers had “caught up” quickly to the more experienced members of the class.

A commonly cited advantage of individual programming, especially early in the semester, was that students felt they learned more effectively when they had to work everything out on their own (30% in Week 6; 11% in Week 12). This illustrates the value of including both individual and Pair Programming sessions.

4 General Discussion

In the interest of finding teaching interventions that can be used successfully in the earliest weeks of a first programming course, we introduced the Pair Programming methodology into our CS1 paper. Based on previous explorations of the pedagogical use of Pair Programming, we intended to construct pairs on ability level, but wished to do so before any exam or significant project marks would be available. We thus used a holistic, subjective judgement made by classroom instructors based on task performance and work style that reflects an attribute we call “programming confidence”. Results of the first semester show that Pair Programming increases practical lab completion rate significantly for those students who were initially judged as having the lowest confidence.

Initial confidence judgements were not correlated with final course mark. Some (but not all) students who had started with low confidence performed very well in the paper; some (but not all) students who started with high confidence levels failed to achieve a high final mark. Thus low initial confidence in isolation is not an indication of future poor performance. However, inspection of change in confidence during the early weeks does seem to give a better insight into eventual outcome. Specifically, students whose confidence level improved between Weeks 2 and 6 of the paper earned higher final marks, on average, than those whose confidence remained at the same absolute level. Thus, it is apparently

difficult to catch up if you fall behind in the first six weeks of CS1. This finding is in concert with the mathematical model of Robins (2010) which demonstrates that failure to thrive in the earliest weeks can be a significant contributor to low pass rates in CS1. To identify students at risk, perhaps with an eye to providing additional support, it seems productive to watch carefully for students who do not gain confidence with programming even very early in their first course. This identification can possibly be made by careful instructor observation – no elaborate assessment metric is required.

Student feedback regarding the use of Pair Programming was generally positive, with respondents identifying advantages mechanical (not having to wait so long for instructor attention), intellectual (the value of a second viewpoint) and social (an effective way to get to know other class members). Students expressed concern about uneven or incompatible pairings, and the classroom instructors report that it is necessary to watch closely for dysfunctional pairings (for example, where one member of the pair is being too dominant) and intervene when required.

In addition to the observed advantages accruing to students, the classroom teaching staff reported a number of positive consequences of using Pair Programming. These included:

- *Shorter waiting times:* Our Programming 1 paper is taught in groups of up to 23 students at a time. During practical work time, classroom instructors move about the room answering questions or offering assistance when students are not progressing. In the first weeks of CS1 when most students have very little idea of how to program, this can be a taxing process for instructors. At our institution we have recently begun assigning two instructors to Programming 1 simply to reduce student wait times. This unfortunately imposes a staffing burden that can be very difficult to manage. With the introduction of Pair Programming, instructors notice a significant reduction in “students waiting with their hands up”. Partially, this is because each instructor intervention now covers two students, but more positively, even novice students, when working with a partner, seem to be able to progress more consistently. As the students frequently observed, two heads are indeed better than one.

- *Increased Engagement:* An historical problem for more experienced students in Programming 1 has been lack of engagement. This is a particular issue for those students who have previously failed the paper, and are repeating it. For these students, the earliest weeks can seem rather pointless. Instructors noted, however, that when working with another more experienced student, repeaters and students with some other prior experience were much more engaged than in previous years. The opportunity to discuss the work with a student of similar level and to perhaps share interesting approaches or possible extensions of the exercises, made the early weeks much more rewarding for students at the top end.

- *Increased Motivation and Performance:* Each set of practical tasks contains one or more “challenge problems”, optional exercises of greater difficulty. The instructors note that students are more likely to attempt

the optional challenge exercises during the Pair Programming practicals than during the individual practicals. This may be a reflection of the confidence obtained from knowing one has a partner to help out on a difficult problem, and/or the desire to perform well when working with another student. Interestingly, the same “striving for excellence” was observed in the major individual project assignment where an unusually high number of students attempted extra credit work, in contrast to previous years.

- *Social Dynamic*: The change which the instructors find the most compelling argument for continuing to use Pair Programming is not directly related to programming performance, but is a generally increased sense of community among the students. Compared to previous years, students are more likely to offer help to each other even in individual labs. Students are more likely to discuss individual assignments and ask for feedback. The general sense of camaraderie and inclusion is higher.

It should be noted that our department has recently introduced a number of other policies that might have contributed to this increased sense of community. In 2012 we have appointed a dedicated first year coordinator responsible for pastoral care of new students, we have established a student common room, built a school Facebook page and increased orientation activities for first year students. All of these probably contribute to the social cohesion seen in Programming 1. However, classroom instructors note that in the specific context of their classroom, they saw social relationships develop during Pair Programming which then grew to include other classroom activities.

In summary, we have found Pair Programming to be a valuable technique from the earliest days of CS1 when students at the same level of programming confidence, as judged by in-class teaching staff, work together. In coming semesters we will continue to introduce Pair Programming early in CS1, and also to incorporate it into our more senior programming papers. With wide-ranging benefits to both students and teaching staff, we see Pair Programming as an essential tool in successful programming education.

5 References

Bennedsen, J. and Caspersen, M.E. (2007): Failure rates in introductory programming. *ACM SIGSCE Bulletin*, **39**(2):32-36.

Cliburn, D. (2003): Experiences with pair programming at a small college. *Journal of Computing Sciences in Colleges*, **19**(1):20-29.

Hanks, B. (2005): Student performance in CS1 with distributed pair programming. *ACM SIGSCE Bulletin*, **37**(3):316-320.

Katira, N., Williams, L. and Osborne, J. (2005): Towards increasing the compatibility of student pair programmers, *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, USA, pp. 625-626.

McDowell, C., Hanks, B., and Werner, L. (2003) Experimenting with pair programming in the

classroom. *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, Thessaloniki, Greece, pp. 60-64.

Mendes, E., Al-Fakhri, L., and Luxton-Reilly, A. (2005): Investigating pair-programming in a 2nd-year software development and design computer science course. *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, Thessaloniki, Greece, pp. 296-300.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, 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, *ACM SIGCSE Bulletin*, **33**(4):125-180.

McDowell, C., Werner, L., Bullock, H., and Fernald, J. (2006): Pair programming improves student retention, confidence, and program quality. *Communications of the ACM* **49**(8):90-95.

Radermacher, A. and Walia, G. (2011): Investigating the effective implementation of pair programming: An empirical investigation, *Proceedings of the 42nd ACM technical symposium on Computer science education*, Dallas, Texas, USA, pp. 655-660.

Robins, A. (2010): Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education*, **20**(1): 37-71.

Thomas, L., Ratcliffe, M., and Robertson, A. (2003): Code warriors and code-aphobes: a study in attitude and pair programming, *ACM SIGCSE Bulletin*, **35**(1):363-367.

Williams, L. (2007): Lessons learned from seven years of pair programming at North Carolina State University. *SIGSCE Bulletin*, **39**(4):79-83.

Distractions in Programming Environments

Raina Mason

Southern Cross University
raina.mason@scu.edu.au

Graham Cooper

Southern Cross University
graham.cooper@scu.edu.au

Abstract

A workshop for teaching introductory programming using Lego Mindstorms NXT presented students with either a ‘complete’ or ‘subset’ form of user interface, both of which are pre-packaged with the application. The learning activities presented to all students only made use of the functionality contained within the subset interface and students presented with the complete interface only made use of the functionality associated with the subset version. Despite no reference to, or use being made of, the extended functionality of the complete interface, students undertaking activities in this mode reported higher levels of difficulty associated with learning programming and performed poorly on a test of programming concepts compared to students presented with the subset form of the interface. Results are explained in terms of Cognitive Load Theory, in particular, redundancy of information. Implications to the selection of programming languages and environments for teaching introductory programming are discussed.

Keywords: Cognitive load, instructional design, introductory programming, Lego Mindstorms.

1 Introduction

Teaching programming is difficult and often unsuccessful (McCracken et al., 2001; Denning and McGettrick, 2005; Ma et al., 2007). This difficulty may be partially caused by the necessity of learning several interacting concepts all at the same time, such as understanding the problem statement, constructing algorithms, navigating syntax rules, semantics, problem solving, navigating a programming interface and compiling and executing a program (Jenkins, 2002).

Regardless of the programming language being used or other aspects of learning programming such as the mechanics of compiling, students must learn how to construct algorithms using the three core structures of sequence, selection and iteration (Dijkstra, 1972). The primary conduit for teaching and learning these three core structures is the user interface, environment and programming language.

Introductory programming courses have many languages and environments from which to choose. There has been debate (Kolling, 1999; Kelleher and Pausch, 2005) regarding the extent to which simplified learning environments may be beneficial in this context. Some

have argued, and developed simplified environments, for pedagogical reasons, but little direct evidence for the reasons behind their success in this purpose have been demonstrated. In contrast, some introductory programming course instructors have contended (see Mason et al. 2012) that introductory programming courses should utilise “professional” level applications as these are used in industry, and that there is no benefit to be gained in requiring students to learn an additional language and environment as a pathway to such professional languages and environments.

The current paper argues that the relatively high level of complexity inherent within most programming languages and environments acts as an impediment to understanding and learning the three core structures of sequence, selection and iteration. Although arguments have been offered for the effectiveness of simpler environments based upon student motivation, visual aspects of the interface and games oriented tasks (Kolling and Henriksen, 2005; Boisvert, 2006; Gomes and Mendes, 2007; Hundhausen et al., 2009) such environments have not been analysed through the lens of cognitive learning theories.

Theories of cognition such as Cognitive Load Theory (Sweller 1999) have been used to develop successful principles of instructional design in other complex domains (for example, see Van Merriënboer et al., 2006). Cognitive Load Theory has previously demonstrated the negative impact that can accrue from split attention (Tarmizi and Sweller, 1988) and redundant information (Mayer et al., 2001). Such studies have typically required students to attend to the ‘complete’ set of information presented. In the context of learning programming there are elements of information that are presented on screen – within the environment – that are irrelevant to the task(s) at hand. Even though such elements may reside outside of the task activities undertaken by students, this paper argues that these may effectively become a source of split attention or redundancy through tacit distraction and thus impede learning.

The primary purpose of the current research is to explore the potential benefits of using simplified languages and environments as pathways to more positive learning experiences, increased self-efficacy with respect to programming, and deeper understanding and transfer of acquired concepts to other computer programming languages.

2 Methodology

2.1 Participants

The experiment was conducted as part of an “IT Careers Day” at a private high school in regional Queensland, with 32 students in Year 7, aged 11 to 13 years. Students participating in the day completed an introductory

programming workshop with Lego Mindstorms NXT robotics (The LEGO Group 2009) followed by a Mindstorms workshop test, then attended a normal school period of class-work. After lunch they participated in an IT careers information session followed by another introductory programming workshop involving the use of the Alice programming environment (Carnegie Mellon University 2006).

2.2 Mindstorms Robots

The Lego Mindstorms NXT kits enable students to build and program robots of their own design. Each kit consists of a central controller “brick” with processing hardware and software, touch, sound, light and ultrasonic sensors, and several servo motors, coupled with ‘technics’-style Lego pieces and gears to enable building of several forms of robots. These robots can respond to input through the sensors and can show output by sound, visual display and movement. The robots are programmed using the Mindstorms NXT software, using a USB cable connection to a PC.

The Mindstorms NXT programming environment is highly visual. Programs are created by dragging programming “blocks” – which are presented as icons representing their functionality - to a timeline, and then setting properties of each block by typing in values or selecting options. The programs are executed in sequence along the timeline. More complex programs can be constructed by using loop and switch/decision structures, as well as ‘wait’ blocks which can be likened to event handling in more traditional languages.

2.3 Design of Workshops

There are two modes of interface presentation available for the Mindstorms software. A “subset” version of the interface presents a truncated set of icon blocks, which are sufficient to build many programs, but lack functionality for tasks such as data storage, data retrieval, mathematical calculations and more specialized functionality. A more “complete” version of the interface presents a greater number of icon blocks representing a greater range of programming tasks, and these are organized into a menu-submenu format.

It was hypothesised that students given either interface would experience an increased knowledge of IT, increased intent to pursue IT as a career, decreased perceived difficulty of programming and increased self-efficacy in programming. The students presented with the subset version of the interface, however, were hypothesised to experience and report amplified effects, resulting in higher self-efficacy and lower levels of the perceived difficulty of programming, than those presented with the more complete interface. It was further hypothesized that students presented with the subset interface would outperform those who received the complete interface on subsequent knowledge tests as measured by both time and score. It was also hypothesised that students who received the subset interface would be more able to transfer their newly acquired (general) knowledge and skills in computer

programming to the Alice computer programming environment.

For these reasons, each student was asked about their knowledge and attitudes towards IT and programming before the IT careers day and at the end of the day, after the Alice workshop. Each student was also asked about the conscious cognitive load which they experienced during the workshop and was given a performance test directly after the Mindstorms workshop. The sequence of activities is presented in Figure 1.

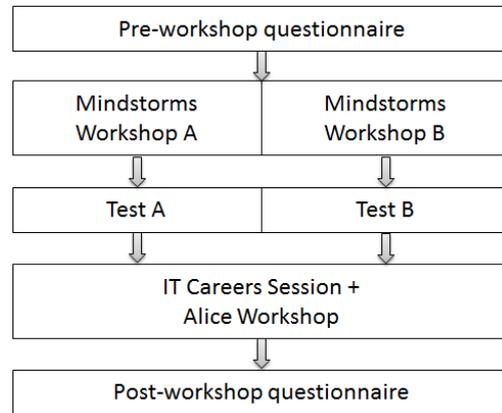


Figure 1: Sequence of Activities

2.4 Instructional Design

Each workshop involved students working either alone or in pairs at one PC computer with one shared robot (built into a humanoid form). There were the same number of pairs and single working students in each group. The instructor introduced the participants to the physical robots and the Mindstorms NXT software and then worked through a series of programming activities with the students. Each programming activity consisted of the instructor demonstrating and describing a small worked example on a large smartboard. The participants then replicated the example using the software, downloaded their solution to the robots and ran their programs. After each student or pair of students completed the activity successfully, the next worked example was demonstrated. At the end of the workshop, students were given time to create their own more complex programs.

The approach used was designed to reduce extraneous cognitive load, and thus facilitate effective learning. Worked examples were used because worked examples have been proven to be a more effective teaching approach than problem solving for novices in technical domains (for examples see Sweller and Cooper, 1985; Cooper and Sweller, 1987; Zhu and Simon, 1987; Paas, 1992). The instructor used verbal explanations at the same time as pictures and processes were demonstrated on the screen, in accordance with the modality principle (Mousavi et al., 1995; Tindall-Ford et al., 1997). The explanations of each step were provided at the same time as the on-screen demonstrations, to reduce cognitive load caused by the split-attention effect (Chandler and Sweller, 1991; Mayer and Anderson, 1991). The instructor also followed a pre-defined script designed using the segmentation principle (Mayer & Chandler 2001) which advocates delivering content in small learner-paced

segments of delivery, moving from simple examples to more complex ones.

The sequence of activities included using the programming environment, simple block use and setting of properties, sequence, looping, events (sensor triggers) and more complex combinations of these concepts. Due to workshop time constraints, decision structures were omitted.

2.5 Treatment Groups

2.5.1 Interfaces

As previously described, the Mindstorms NXT software uses programming blocks to build programs using drag and drop placement on a graphical timeline. The default palette of blocks includes the most commonly used programming blocks and is the first palette available when a new program is created for the first time. Most programming blocks are available on the left of the screen, with one group of programming blocks situated in a slide-out sub-palette. This interface was designated as the Subset interface (see Figure 2).

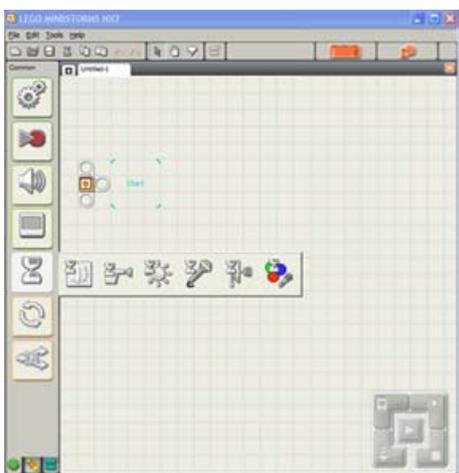


Figure 2: Mindstorms Subset Interface

The software can also be configured so that a more comprehensive set of programming blocks are available when a new program is created. This more comprehensive palette includes the common blocks available in the Subset interface and repeats these blocks and adds new blocks in several sub-palettes accessed through side buttons/icons (for example see Figure 3).

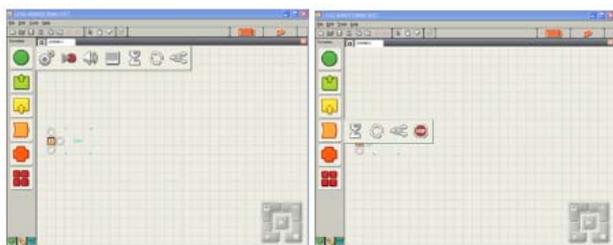


Figure 3: Mindstorms Complete interface

The user has a choice of more programming blocks in this interface, from now referred to as the Complete interface to distinguish it from the Subset interface. The buttons are the same size in each interface and buttons

that have the same functionality have the same appearance.

2.5.2 Distractors

Cognitive Load Theory has previously demonstrated that redundant information may interfere with learning (Mayer et al., 2001). In the context of the activities undertaken by students the Complete interface presented icon blocks that were unnecessary to the tasks to be undertaken. These additional icon blocks were never referenced in any instructions or activities. It is hypothesised that their mere presence on screen would act as a form of tacit distractor. It was hypothesised that this would detract from student learning of the interface. It was further hypothesised that this tacit distraction would also reduce participants learning of the core underlying computer programming concepts and procedures being presented within the learning activities. It was hypothesised that this would lead to lower levels of self-efficacy. Finally, and most importantly, it was hypothesised that such tacit distraction would also reduce the scope to transfer newly learnt concepts and procedures to other programming environments.

2.5.3 Groups

Students were divided into two groups/workshops with roughly equal numbers of each gender (Group Subset: 9 males, 8 females; Group Complete: 7 males, 8 females) in each group. Group Subset used the subset interface throughout their workshop. Group Complete used the complete interface throughout their workshop. Both groups used the same programming blocks, and completed the same activities in the same time. Even though Group Complete had a greater number of blocks available via the palette on-screen, they were not directed to use any of these extra blocks, in any of the worked example activities or free programming time. The extra blocks were not referenced or explained in any way.

2.6 Instruments

Prior to the careers day, each participant completed a questionnaire consisting of demographic questions about age and school year, as well as several 9-point Likert scale questions concerning their level of computer literacy, programming experience, knowledge of IT, intent to pursue a career in IT, perception of the difficulty of programming, and confidence with programming.

After the Mindstorms workshop, the participants were given an equivalent test dependent on their treatment group. The timed, written tests were designed to test recall of the purpose of various programming blocks, the building of schema about the interface used, near transfer of programming construct concepts as well as far transfer of concepts such as sequence, looping and events. The test instrument was also used to collect data about the mental effort used in each of the three aspects of completing an activity in the Mindstorms workshop: understanding what needed to be done in each exercise, navigating and using the NXT software, and learning and understanding concepts.

At the conclusion of the careers day, after the Alice workshops, participants completed a post-workshop

questionnaire which again asked about their level of knowledge of IT, intent to pursue a career in IT, perception of the difficulty of programming and confidence with programming. Participants were also asked about their level of interest in programming with Mindstorms and with Alice, and how difficult they found each of the programming workshops.

3 Results

3.1 Homogeneity of Groups

There was no significant difference between groups in participant age, computer literacy, programming experience, knowledge of IT and intent to pursue a career in IT between Group Subset and Group Complete, before the workshop. A t-test on age returned no significant difference ($\text{Mean}_{\text{Subset}} - \text{Mean}_{\text{Complete}} = -0.08$; $t = -0.3$; $df = 24$; $p = 0.77$, and Mann-Whitney U tests on computing skill, programming experience, knowledge of IT and intent to pursue a career in IT all returned no significant difference between the two groups at the 0.05 level (see Table 1). (It should be noted that an alpha level of $p = 0.05$ is used for all tests reported in this paper)

	U_A	z	p
Computer Literacy	87	0.1	0.46
Prog. Experience	97	-0.62	0.27
IT Knowledge	79	0.26	0.40
Career Intent	83.5	0.03	0.49

Table 1: Homogeneity of Groups

3.2 Career Aspirations and IT Knowledge

Although there were 17 participants in Group Subset and 15 participants in Group Complete for the actual Mindstorms workshops and test, not all participants completed both the pre- and post-workshop questionnaires, as some students did not continue to the Alice workshops due to timetable clashes with other classes. The answers to the pre- and post-workshop questionnaires of the 12 participants in each group who did complete both questionnaires were analysed using the Wilcoxon Signed Ranks test. The median, minimum and maximum scores for each are given in Table 2 below.

It was expected that all participants would be more knowledgeable and more likely to consider a career in IT after the workshops, as a result of the positive experiences, and career information given to them. This was obtained with a significant difference in the perceived IT knowledge of participants after the workshop than before the workshop [Wilcoxon Signed Rank test: $n = 24$, $W = 179$, $N_{s/r} = 21$, $z = 3.1$, $p = 0.001$] and a significant difference in participants' intent to consider a career in IT after the workshop than before the workshop [Wilcoxon Signed Rank test: $n = 24$, $W = 128$, $N_{s/r} = 19$, $z = 2.57$, $p = 0.005$].

	n	median	mode	min	max
Knowledge - pre	24	5	5	1	7
Knowledge - post	24	6	7	2	9
Career - pre	24	4.5	1	1	9
Career - post	24	5.5	7	1	9

Table 2: Results of IT knowledge/Career intent

3.3 Programming Difficulty and Self-Efficacy

In both questionnaires, participants were asked to indicate how much they agreed with the statement "I think programming generally is difficult" on a 9 point Likert scale (where 1 = 'no way' and 9 = 'totally!'). It was anticipated that participants would display a decrease in the perceived level of difficulty of programming after the Careers Day when compared to before the Careers Day. A significant decrease was obtained in participants' perception of the difficulty of programming, from before the workshops to after the workshops [Wilcoxon Signed Rank test: $n = 24$, $W = 128$, $N_{s/r} = 19$, $z = 2.57$, $p = 0.005$]. The median, minimum and maximum scores for perceived level of difficulty of programming are shown in Table 3 below.

	n	median	mode	min	max
Pre-workshop	24	5	5	1	9
Post-workshop	24	4	5	1	7

Table 3: Measures of Difficulty of Programming

It was also hypothesised that Group Subset may have a greater shift in perception of difficulty in the direction of "easier" than Group Complete, who were presented with the fuller interface in the Mindstorms workshops.

Further analysis of the pre- and post-workshop answers from the 12 participants in Group Subset and 12 participants in Group Complete who completed the Mindstorms workshops and the Alice workshops and who completed both questionnaires were then analysed individually using Wilcoxon Signed Rank tests. The results are shown in Table 2.

	n	W	$N_{s/r}$	z	p
Subset	12	-55	11	-2.42	0.008
Complete	12	-29	10	-1.45	0.07

Table 4: Difficulty of Programming

Group Subset had a significantly different perception of the difficulty of programming after the workshop compared to before the workshop, in the direction of 'easier'. Although Group Complete displayed shifts downwards, this group did not experience a significant change in their perception of the difficulty of programming.

In pre- and post-workshop questionnaires, participants were asked to indicate how much they agreed with the statement "I feel confident with programming" on a 9 point Likert scale (where 1 = 'no way' and 9 = 'totally!'). Previous workshop results (Mason et al., 2011a, 2011b)

indicated that it was likely that participants would report increased self-efficacy in computer programming after the workshops. However it was anticipated that Group Subset may have a greater increase in self-efficacy in programming than Group Complete.

Answers from both groups were analysed together using Wilcoxon Signed Rank tests and participant's self-efficacy was found to be significantly higher after the workshops than before the workshops [$n = 24$, $W = 119$, $N_{s/r} = 18$, $z = 2.58$, $p = 0.005$]. The median, minimum and maximum scores for participants' self-efficacy before and after the workshops are shown below in Table 5.

	n	median	mode	min	max
Pre-workshop	24	5	5	1	9
Post-workshop	24	7	9	2	9

Table 5: Measures of self-efficacy

Pre- and post-workshop measures were then analysed separately for the subset and complete interface groups using Wilcoxon Signed Rank tests and the results are shown in Table 6 below:

	n	W	$N_{s/r}$	p
Subset	12	42	9	$p < 0.01$
Complete	12	18	9	$p > 0.05$

Table 6: Self-Efficacy in Programming

Group Subset had a significantly higher self-efficacy in programming after the workshop than before the workshop. Although Group Complete displayed some shifts upwards in self-efficacy, this group did not experience a significant change.

3.4 Test Performance

3.4.1 Test Completion Time

It was hypothesised that if the different interface had an effect on learning then participants from Group Subset would take less time to complete the test than participants in Group Complete. The times from the 17 participants in Group Subset and 15 participants in Group Complete were compared using a t-test. Participants from Group Subset were found to take significantly less time to complete the test than Group Complete [$Mean_{Subset} - Mean_{Complete} = -150$ seconds; $t = -1.91$, $df = 30$, $p = 0.03$].

3.4.2 Test Score

There was a total maximum possible mark of 17 for the Mindstorms test. The sequence of questions and associated marks were allocated as follows:

- 1 mark each for correct description of the purpose of programming blocks [total 3 marks];
- 1 mark each for correct placement of programming block on blanked interface [total 6 marks];
- 1 mark each for correct multiple choice answers (near transfer) [total 5 marks];
- 1 mark each for far transfer answers [total 3 marks].

The two groups mean total marks, standard deviation, minimum and maximum are shown in Table 7.

	mean	st dev	min	max
Subset	11.03	2.60	6.5	15
Complete	8.83	1.88	6	14

Table 7: Test Score Totals

The total test scores from the 17 participants in Group Subset and 15 participants in Group Complete were compared using a t-test, and participants from Group Subset were found to have a significantly higher test score than Group Complete [$Mean_{Subset} - Mean_{Complete} = 2.20$; $t = 2.71$, $df = 30$, $p = 0.006$].

3.4.3 Interface Schema Acquisition

Students were asked to indicate where six programming blocks were placed on a blanked version of the Mindstorms interface. Each of these six programming blocks had been used in the Mindstorms workshop activities at least twice. It was anticipated that students in Group Subset would more effectively build schema for the positioning of the blocks in the interface over students in Group Complete interface, who would have more of their working memory capacity used in needing to deal with the tacit distractors of the availability of extra blocks – even though those blocks were not being used or referenced in any of the activities.

The maximum score available on this question was 6, if the student placed all of the blocks in the correct positions. The Complete Interface makes several programming blocks available in more than one location, so if a student in Group Complete placed that particular block in any of the correct positions, they received a point for that block.

The two groups mean marks, standard deviation, minimum and maximum are shown in Table 8.

	mean	st dev	min	max
Subset	2.76	1.71	0	5
Complete	1.33	1.18	0	4

Table 8: Interface Schema Score totals

Participants from Group Subset scored significantly higher on rebuilding the interface than participants from Group Complete [t-test: $Mean_{Subset} - Mean_{Complete} = 1.43$, $t = 2.72$, $df = 30$, $p = 0.005$].

3.4.4 Knowledge Acquisition

The maximum score available for the test, *excluding* rebuilding the interface, was 11. The two group scores for the knowledge acquisition part of the test were compared using a t test and although these results were not statistically significant [$Mean_{Subset} - Mean_{Complete} = 0.7647$, $t = +1.36$, $df = 30$, $p = 0.09$], the results trended in the expected direction. Closer inspection of the test scores for Knowledge Acquisition showed that 9 of the 17 participants in Group Subset scored 9 or higher (from a possible 11), while in Group Complete, only 2 of the 15

participants scored 9 or above. This result is statistically significant ($p = 0.02$) using Chi-Square analysis. This indicates that more participants in Group Subset scored highly (at 9 or more marks out of a possible 11 marks) for Knowledge Acquisition than those participants in Group Complete.

3.5 Difficulty of Programming Environments

3.5.1 Mindstorms environment Difficulty

In the post-IT Careers Day questionnaire, participants were asked to complete the statement "Programming robots to do things is .." on a 9 point Likert scale (where 1 = 'really easy' and 9 = 'really difficult'). It was expected that if having the extra (unused) programming blocks available in the Complete interface were having an effect on working memory load while using the interface, then Group Complete would perceive programming robots as more difficult than Group Subset, even though both groups completed the same activities with the same programming blocks.

The responses of the 15 participants in Group Subset and 14 participants in Group Complete who completed the Mindstorms workshops and completed the post-workshop questionnaire were analysed for differences using a Mann-Whitney U Test. Participants in Group Complete found programming with the Mindstorms Robots significantly more difficult than participants in Group Subset [$U_A = 144$, $z = 1.66$, $p = 0.049$]. The median, minimum and maximum scores are given below in Table 9.

	n	median	mode	min	max
Group Subset	15	2	1	1	5
Group Complete	14	3.5	1	1	9

Table 9: Mindstorms Difficulty

This was an expected result. The presence of the extra (unused) programming blocks was interfering with the attentional process for the students in Group Complete.

3.5.2 Alice Environment Difficulty

In the post-IT Careers Day questionnaire, participants were also asked to complete a similar statement about the difficulty of programming with Alice - "Programming with Alice 3D worlds is .." on a 9 point Likert scale (where 1 = 'really easy' and 9 = 'really difficult').

Note that both Group Subset and Group Complete were mixed in a common Alice programming session in the afternoon, after completing the Mindstorms workshops.

Both groups completed the same Alice workshop activities, with the same Alice interface, and were asked afterwards about the difficulty of programming in Alice. As all participants were in the same Alice programming workshop, the only variable was the Mindstorms group in which each participant had participated. It was hypothesised that the difference in difficulty experienced in the Mindstorms workshops as a result of having the subset interface for participants in Group Subset, compared to those in Group Complete, would have a

positive transfer effect to the perceived difficulty of Alice programming. That is, participants that had experienced the Subset interface for Mindstorms, would perceive programming in Alice as less difficult than those who had experienced the Complete Mindstorms Interface.

The responses of the 15 participants in Group Subset and 14 participants in Group Complete who completed the Mindstorms workshops and completed the post-questionnaire after the Alice workshop were analysed for differences using a Mann-Whitney U Test. The novice programming participants in Group Complete found programming with *Alice* significantly more difficult than novice programming participants in Group Subset [$U_A = 158$, $z = 2.29$, $p = 0.01$]. The median, minimum and maximum scores are given below in Table 10.

	n	median	mode	min	max
Group Subset	15	1.5	1	1	7
Group Complete	14	5	5	1	9

Table 10: Alice Difficulty

This result shows a transfer effect from the Mindstorms workshop to the Alice workshop based upon the nature of the Mindstorms interface that participants had received. This is discussed more in Part 4: Discussion.

3.5.3 Cognitive Load

It was expected that participants in Group Subset would report lower conscious mental effort (cognitive load) measures in "navigating and using the Mindstorms NXT software" and their overall patterns of mental effort would differ from participants in Group Complete. The performance measures indicate that learning was more effective for Group Subset. We had hypothesized that this would be the result of lower cognitive load being imposed upon Group Subset. Although there were trends present in the direction of lower reported cognitive load in Group Subset, there were no significant differences between groups.

4 Discussion

The benefits of the Careers Day intervention for both girls and boys who were novices to programming were obvious, with positive shifts in IT knowledge, IT career aspirations and self-efficacy in programming, and lowered perception of the difficulty of programming, from before the workshops to after the workshops.

The effect of the treatment and differences between groups became evident on analysing their performance in the Mindstorms test. On each of the four chosen measures (Test Completion Time, Total Test Score, Interface Schema Acquisition Score and Knowledge Acquisition Score), Group Subset outperformed Group Complete. There was a demonstrated advantage for the students who were given the simpler interface with fewer options, even though the extra options for Group Complete were not used.

It was expected that if the students in Group Complete were experiencing higher cognitive load on working memory during the Mindstorms workshop than Group

Subset as a result of the extra options on screen, they would experience the task of programming in the Mindstorms environment as more difficult, compared to Group Subset. Group Complete did find programming in the Mindstorms environment significantly more difficult than Group Subset which supports this hypothesis.

There were, however, no significant differences observed between groups on reported cognitive load measures. This may be due to lack of sensitivity in the Likert test questions used with this participant pool, or may represent another dynamic associated with a form of tacit distraction. This issue warrants further research.

Both treatment groups then participated in the same Alice workshop, at the same time, with the same materials and instruction. There was a significant transfer effect, with those participants in Mindstorms Group Subset finding Alice programming significantly easier than participants who had been in Group Complete.

This is a potentially critical outcome. At the heart of all computer programming languages and environments are the core, fundamental concept blocks that enable the design and development of suitable algorithms. The results obtained in the present study argue that the schemas for these underlying mental representations may be facilitated through the use of entry-level computer programming environments specifically designed to cater for novices.

These results have implications for the often-debated question about whether it is better to introduce introductory programming students to an Integrated Development Environment (IDE) that is used in industry first, or to introduce students to programming using a teaching environment first, and then move on to an industry standard IDE later. Instructors that are in favour of using an industry standard environment for introductory programming courses often point to the extra effort of teaching (and for students, in learning) two environments (de Raadt et al., 2002; Mason et al., 2012).

The results of this research indicate that for novices to programming, having extra options available in the environment - *even if they are not used or referenced* - hinders learning (reduces performance) and causes the students to perceive programming in both that environment and subsequent environments as more difficult. The results of this study indicate that novice students benefited from a simplified first-programming environment. This facilitated learning of core programming constructs as measured on test performance and also transferred to a second programming environment as measured by reported perceptions of programming difficulty.

While the current study was specifically focussed upon novice programmers and programming environments, it is worth noting that relatively many computer applications present users with additional icons and functionality that are redundant (or unnecessary) to their task performance. Such "over-provision" of functionality may be misguided and impede learning of the application.

The current study may have limited scope because it used participants who were school students rather than university students, because the programming environments used were heavily icon based rather than

line code, and because the entire exposure of participants was across a single day, rather than across an entire year (or longer). Nevertheless, the design of the specific instructional materials used, and the broad complex of results obtained, were driven by the application of Cognitive Load Theory to the context of teaching novices some of the basic concepts, structures and processes of computer programming.

Cognitive Load Theory provides an extensive body of empirical studies demonstrating utility in enhancing instructional design in complex areas of learning such as mathematics, science and industry technical applications (Sweller, 1999). The results reported here represent another example of Cognitive Load Theory being usefully applied to enhance the design of instructional materials in an area of high conceptual and task complexity...that of computer programming...and all computer programmers begin their life as programmers...as novices.

5 Thanks

The authors would like to acknowledge the helpfulness and involvement of the school teachers and students who were the participants in this study.

6 References

- Boisvert, C., 2006. Web animation to communicate iterative development. *ACM SIGCSE Bulletin* 38, 173–177.
- Carnegie Mellon University, 2006. What is Alice and what is it good for? [WWW Document]. URL http://www.alice.org/index.php?page=what_is_alice/what_is_alice
- Chandler, P., Sweller, J., 1991. Cognitive Load Theory and the Format of Instruction. *Cognition and Instruction* 8, 293–332.
- Cooper, G., Sweller, J., 1987. Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of Educational Psychology* 79, 347–362.
- de Raadt, M., Watson, R., Toleman, M., 2002. Language trends in introductory programming courses [WWW Document]. *Informing Science + IT Education Conference*. URL <http://proceedings.informingscience.org/IS2002Proceedings/papers/deRaa136Langu.pdf>
- Denning, P., McGettrick, A., 2005. Recentring Computer Science. *Communications of the ACM* 48, 15–19.
- Dijkstra, E.W., 1972. Notes on Structured Programming, in: *Structured Programming*. Academic Press, New York, NY, pp. 1–82.
- Gomes, A., Mendes, A.J., 2007. An environment to improve programming education, in: *Proceedings of the 2007 International Conference on Computer Systems and Technologies - CompSysTech '07*. ACM Press, Bulgaria, pp. Article 88, 6 pages.
- Hundhausen, C.D., Farley, S.F., Brown, J.L., 2009. Can direct manipulation lower the barriers to computer programming and promote transfer of training? *ACM*

- Transactions on Computer-Human Interaction 16, 1–40.
- Jenkins, T., 2002. On the difficulty of learning to program, in: Proceedings of the 3rd Annual Conference of the LTSN-ICS. Loughborough, Ireland, pp. 53–58.
- Kelleher, C., Pausch, R., 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)* 37, 83–137.
- Kolling, M., 1999. The problem of teaching object-oriented programming, part 2: Environments. *Journal of Object-Oriented Programming* 11, 6–12.
- Kolling, M., Henriksen, P., 2005. Game programming in introductory courses with direct state manipulation, in: Proceedings of ITiSCE'05. ACM Press, Caparica, Portugal, pp. 59–63.
- Ma, L., Ferguson, J., Roper, M., Wood, M., 2007. Investigating the viability of mental models held by novice programmers, in: Proceedings of the 38th Technical Symposium on Computer Science Education. ACM Press, pp. 499–503.
- Mason, R., Cooper, G., Comber, T., 2011a. Girls get IT. *ACM Inroads* 2, 71–77.
- Mason, R., Cooper, G., Comber, T., 2011b. It's (no longer) a remote chance for girls in IT, in: Proceedings of the 1st International Australasian Conference on Enabling Access to Higher Education 2011. University of South Australia, Adelaide, Australia, pp. 310–321.
- Mason, R., Cooper, G., de Raadt, M., 2012. Trends in Introductory Programming Courses in Australian Universities – Languages, Environments and Pedagogy, in: de Raadt, M., Carbone, A. (Eds.), Proceedings of the Fourteenth Australasian Computing Education Conference (ACE2012). Australian Computer Society, Inc., Melbourne, Australia, pp. 33–42.
- Mayer, R.E., Anderson, R.B., 1991. Animations need narrations: An experimental test of a dual-coding hypothesis. *Journal of Educational Psychology* 83, 484–490.
- Mayer, R.E., Chandler, P., 2001. When learning is just a click away: Does simple user interaction foster deeper understanding of multimedia messages? *Journal of Educational Psychology* 93, 390–397.
- Mayer, R.E., Heiser, J., Lonn, S., 2001. Cognitive constraints on multimedia learning: When presenting more material results in less understanding. *Journal of Educational Psychology* 93, 187–198.
- McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.-D., Laxer, C., Thomas, L., Utting, I., Wilusz, T., 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin* 33, 125–180.
- Mousavi, S.Y., Low, R., Sweller, J., 1995. Reducing cognitive load by mixing auditory and visual presentation modes. *Journal of Educational Psychology* 87, 319–334.
- Paas, F., 1992. Training strategies for attaining transfer of problem-solving skills in statistics: A cognitive-load approach. *Journal of Educational Psychology* 84, 429–434.
- Sweller, J., 1999. Instructional design in technical areas. The Australian Council for Educational Research Ltd, Camberwell, VIC.
- Sweller, J., Cooper, G., 1985. The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra. *Cognition and Instruction* 2, 59–89.
- Tarmizi, R.A., Sweller, J., 1988. Guidance during mathematical problem solving. *Journal of Educational Psychology* 80, 424–436.
- The LEGO Group, 2009. MINDSTORMS [WWW Document]. URL <http://mindstorms.lego.com/en-us/Default.aspx>
- Tindall-Ford, S., Chandler, P., Sweller, J., 1997. When two sensory modes are better than one. *Journal of Experimental Psychology: Applied* 3, 257–287.
- Van Merriënboer, J.J.G., Kester, L., Paas, F., 2006. Teaching complex rather than simple tasks: balancing intrinsic and germane load to enhance transfer of learning. *Applied Cognitive Psychology* 20, 343–352.
- Zhu, X., Simon, H.A., 1987. Learning mathematics from examples and by doing. *Cognition and Instruction* 4, 137–166.

Identifying career outcomes as the first step in ICT curricula development

Nicole Herbert, Kristy de Salas, Ian Lewis, Mike Cameron-Jones, Winyu Chinthammit,
Julian Dermoudy, Leonie Ellis, Matthew Springer

School of Computing and Information Systems

University of Tasmania

Private Bag 87, Hobart 7001, Tasmania

Nicole.Herbert@utas.edu.au

Abstract

While much advertising for ICT degrees uses career outcomes to market them to potential students, there is little evidence about whether these outcomes have been truly embedded into the curriculum and hence whether they can actually be attained by students. This paper reports on a process to design a University ICT curriculum that is directly informed by the career outcomes relevant to the local and national ICT industry.

Keywords: ICT career outcomes, ICT skills, ICT curriculum, ICT graduates, ICT degree

1 Introduction

It is well known that ICT curricula are in a constant state of flux in response to continuing changes in emerging technology and resources such as staffing levels, student numbers, and funding models. It is often unclear whether specified career outcomes for particular degrees are part of the curriculum development process or just an advertising mechanism. A study undertaken to investigate the drivers of curriculum change (Gruba et al 2004), discovered that change is predominantly driven by outspoken individuals, budgetary constraints, and student demand rather than academic merit and external curricula. In attempts to respond to the constraints and ever-changing technology it is easy to lose sight of the advertised career outcomes as a focus. Calitz et al (2011) stated that academics and students need to acquire a thorough knowledge of ICT career outcomes and that *“universities must link and publish computing programs, linking each program with specific career tracks, indicating specific career specialisation and knowledge.”*

There is little evidence that career outcomes as stated on marketing materials are really attainable by students. Graduate career prospects are one of the major influencing factors when pre-tertiary students (and their parents) are selecting their degree. Babin et al (2010) and Biggers et al (2008) suggested that the main reason for the lack of interest in a career in ICT by pre-tertiary students is that computing is traditionally perceived as asocial, focusing on programming and having limited connections to the outside world. To counter this negative and inaccurate perception, and to promote the future

growth of the industry, it is essential that the career outcomes for modern ICT degrees reflect the myriad of career opportunities now available (ACM 2008) and the curriculum is designed such that graduates can attain the skills for these careers (von Kinsky 2008).

While theoretically linking curricula design closely with career outcomes might be an ideal situation, in practice, tertiary institutions are currently juggling the different demands of local and international students and there has been increased specialisation of programs and a correspondingly large growth in the number of units (subjects) on offer. In 2012, UTAS (2012) has 50 ICT undergraduate units that, although a sizeable offering, is small compared to the undergraduate listings in other universities' 2012 handbooks. For example, Monash (2012) has 120 (code FIT), Swinburne (2012) has 163 (code HET, HIT), and QUT (2012) has 64 (code INB). This is a common problem identified by Henkel and Kogan (1999), who suggest that an emphasis on academic objectives tends not to be coherent but results in a large range of topics for students and will typically include the research interests of staff. Alternatively Henkel and Kogan suggest when emphasis is placed on employment objectives the resulting curricula are more directed and coherent.

While an abundance of units might allow for an abundance of career opportunities, this makes isolating core career outcomes very difficult and therefore also difficult for students to know exactly what units to take to achieve a desired career outcome. Alexander et al (2010) and Nagarajan and Edwards (2008) found that graduates find it very difficult to identify ICT career opportunities that relate to the skills they have developed during their study. Furthermore, this abundance in units, and course specialisations, makes it difficult for industry to determine solely on the basis of a graduate's degree whether they are qualified for a particular career, instead requiring knowledge of specific unit content.

The remaining sections of this paper will describe a process for developing a leading-edge and innovative ICT degree that is directly informed by the career outcomes relevant to the local and national ICT industry. The Australian Computer Society (ACS) provide a process (*what to do*) to guide the development of new curricula (ACS 2011), but not the specific activities to undertake (*how to do it*). This paper focuses on how to perform the first three of seven steps in the ACS process to develop a new ICT degree, namely:

- identify potential ICT roles that could be undertaken by graduates of a given program of study;
- identify the SFIA skills required by professionals in a given ICT career role;
- identify the level of autonomy and responsibility to be developed.

This paper is one of a pair of papers that describe curriculum design using the ACS process. The companion paper (Herbert et al 2013) describes the outcomes of ACS process steps four to six.

There is little literature focussing on *how* to link career outcomes and ICT curricula. This paper describes specific techniques for identifying potential career outcomes, SFIA skills and responsibility levels during an ICT curriculum development effort guided by the ACS process (ACS 2011). As our implementation of our process will also be important to some readers our constraints, resources and outcomes of each phase of the process are included for completeness.

2 Process overview

Figure 1 outlines our four-stage process by which career outcomes are identified and used to guide subsequent curricula design decisions. The Figure also includes a summary of our constraints, resources and outputs to make it easier to follow our implementation of the process throughout the paper. As already mentioned, the process is based on the ACS recommended process (ACS 2011) with additional details on *how* to perform each step. Before commencing the process it is worthwhile identifying any constraints that will impact on decisions

made about career outcomes and skills developed throughout the process.

Our Constraints

The University of Tasmania (UTAS) is the only university within Tasmania, and the School of Computing and Information Systems, as the only ICT School at UTAS, must meet the ICT higher education needs of the ICT industry in Tasmania. In order to maintain the currency and quality of the school’s activities, an external school review was undertaken in 2011. One primary outcome was the recommendation to discontinue the two current undergraduate degrees: a Bachelor of Computing and a Bachelor of Information Systems; and to instead create a new single degree — a Bachelor of ICT. This would be the *only* tertiary ICT degree on offer in Tasmania.

A shrinking staff profile and simultaneous pressure for increased research output across the whole of UTAS resulted in the review panel recommending a reduction of the number of undergraduate unit offerings from fifty to just thirty. We need to develop a coherent curriculum in which every one of the thirty units is achieving its maximum potential by working towards providing graduates with a broad range of ICT skills covering the essential technical and non-technical ICT skills and the professional skills needed to enhance the Tasmanian ICT industry.

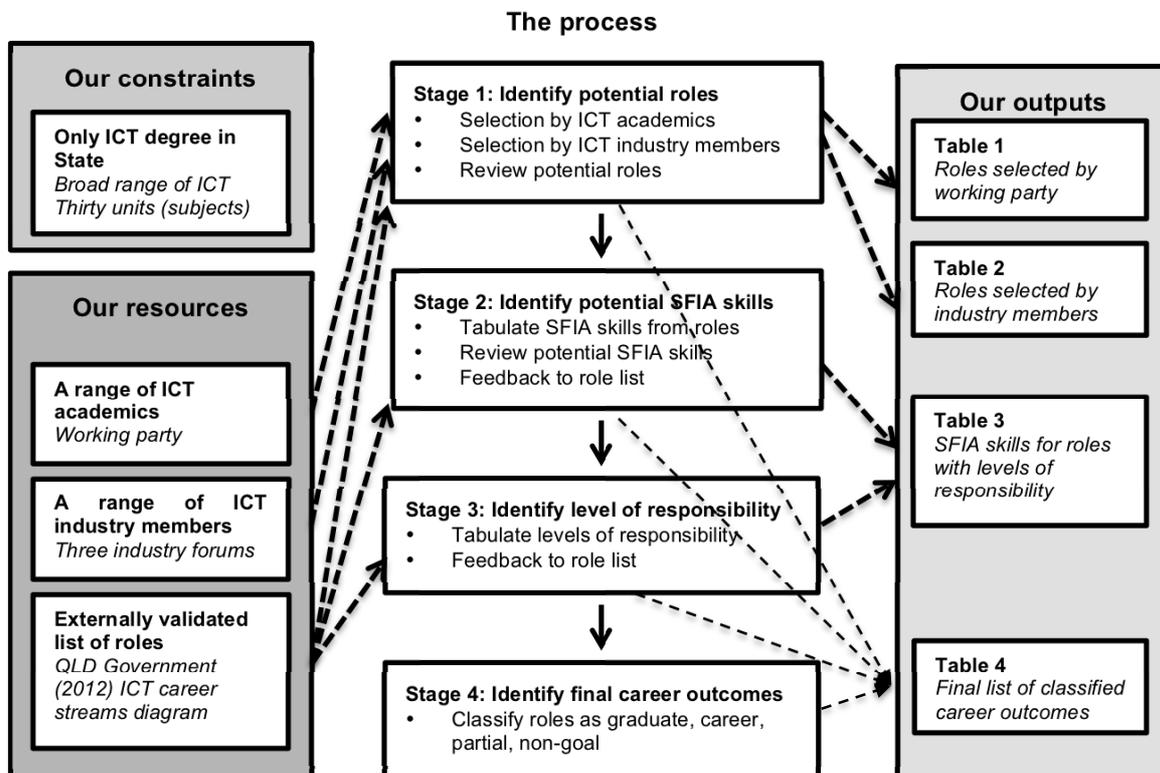


Figure 1: The process to identify career outcomes for an ICT degree

2.1 Stage 1: Identify potential roles

Our initial investigation into ICT degrees throughout Australia indicated that degrees aim to produce graduates qualified for a range of ICT careers, and although there are some common career outcomes, most are quite different in their emphasis. ICT is constantly changing and new technology is continuously emerging and as a result career titles and definitions are changing. Our investigation indicated there appears to be no nationally recognised standard set of career titles and definitions that are used or maintained.

If career outcomes are to be achieved, they must be embedded into design. The first essential step must be to identify an externally validated set of ICT career definitions that covers a broad range of ICT careers. External validation limits the “influence of outspoken individuals” (Gruba et al 2004). At the conclusion of this stage a number of career outcomes is required to create a degree that will meet its objectives as well as give graduates options.

Our Resource — ICT career streams diagram

The Queensland Government Chief Information Office (QLD Government 2012) developed an ICT career streams diagram in 2006. It is maintained to keep it current, and was last updated in 2012. This diagram (shown in Figure 2) identifies four different career streams and 55 key ICT roles. The online version of the diagram is interactive and selecting a role will take the user to further information that clearly defines the role and identifies the SFIA (Skills Foundation in the Information Age) skill set required to perform the role along with the level of responsibility (SFIA 2012).

Given our constraint of only thirty units it was necessary to identify a subset of the 55 roles that would be career outcomes for our new degree. At the conclusion of this stage a broad range of career outcomes was required in order to create a non-specialised ICT degree that would have wide appeal.

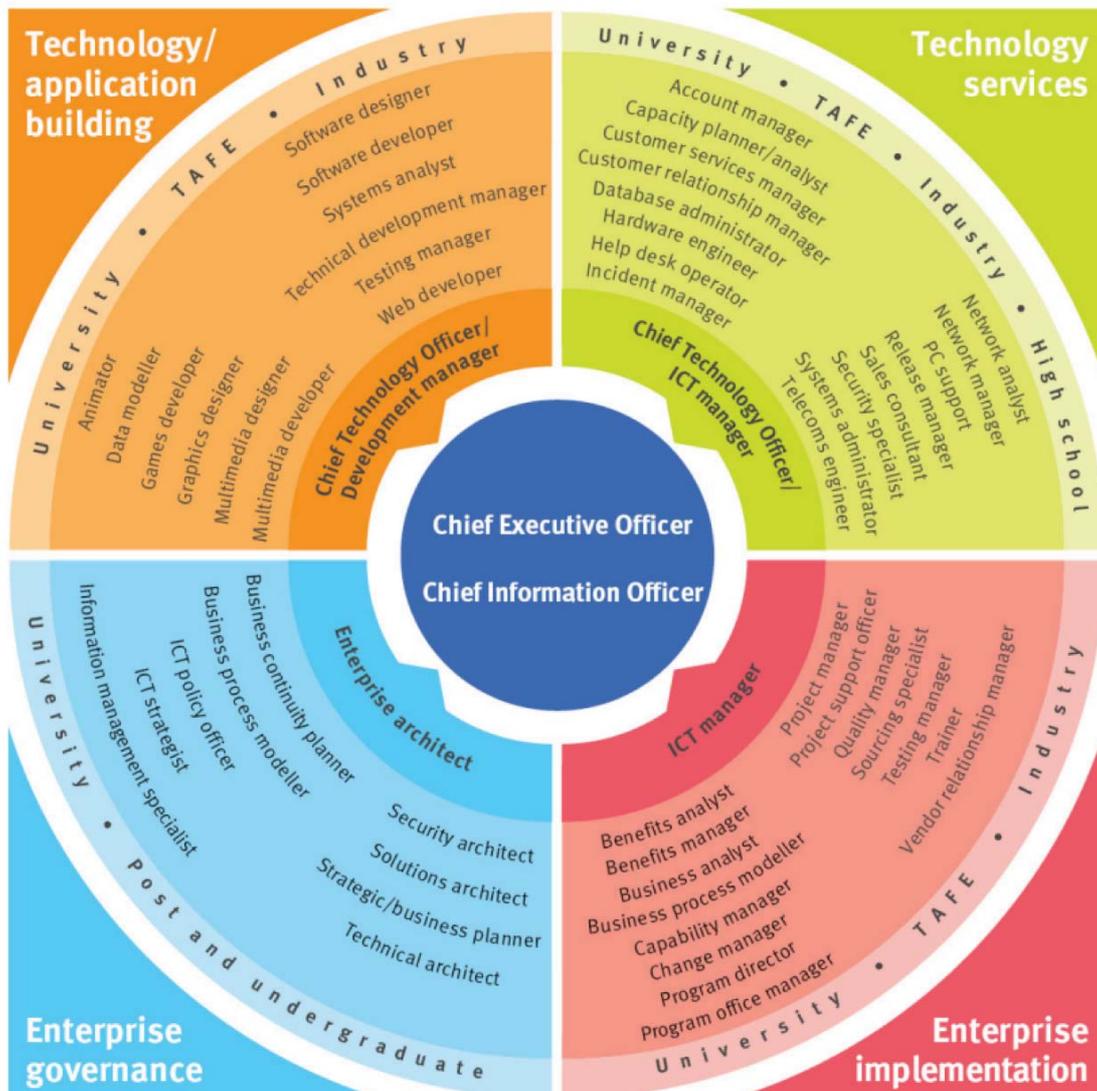


Figure 2: ICT Career Streams, included with permission of QLD Government (2012)

There is a difference between the roles a graduate could be fully qualified for on graduation and those careers that they might aspire to over time. It is useful, not only for accurate marketing to potential students and their parents but also for the latter stages of the process, to be able to distinguish between the different roles available: graduate roles that students can perform when they enter the workforce, career roles they might eventually achieve after a few years of experience, or partially-qualified roles that they might not develop all the skills required during an undergraduate degree and require further study. As a result, to guide the process of identifying relevant career outcomes it is necessary to preface any selection process with an identification of the extent of qualification:

- *Fully* — indicates students should be fully qualified for graduate entry in this role. Some short specific training maybe required, but graduates are expected to be fully capable of performing this role in a business within six months. Unit content should be focussed towards this role.
- *Partial* — indicates students should have some useful skills for this role but not all. There maybe content that should not be supplied at university undergraduate level. It could be supplied by another organisation or a postgraduate degree.
- *No* — indicates this is not a role to aim for with the degree (whether it is achieved by skill overlap is irrelevant at this time).
- *Unsure* — indicates the reviewer was undecided.

Unless an institution has unlimited resources or unless a small set of career definitions was chosen to begin with, it is necessary to identify a subset of the roles that are relevant for the new degree. It is recommended that input from all stakeholders is sought but at the very least a two-stage process is recommended:

- selection by ICT academics; and
- selection by ICT industry members.

2.1.1 ICT academics

Academics that will be implementing the new degree should be involved in the identification process. Inviting academics who will be involved in the implementation of the new curriculum to be involved in the design from the outset builds a sense of ownership that will facilitate change (Elizondo-Montemayor 2008). To ensure that the new degree is not heavily influenced by any one individual a range of staff should be invited to identify the roles they deem relevant from the externally validated set of career definitions.

Our Resource — Working party

A working party was formed consisting of eight academics, heavily interested in teaching and learning with a variety of different characteristics and backgrounds: three from the Launceston Campus, five from the Hobart campus; three primarily from the Information Systems discipline, five primarily Computer Science; and three being female and five male.

Our Output — Working party

Only a small number of roles, 8 (out of 55), received 5 (out of 8) or more *Fully* votes. There were 16 roles that received at least 75% (6 out of 8) of the votes when combining the *Fully* and *Partial* votes. The results are shown in Table 1. The careers that are different to the ICT industry member responses are shown in italics.

2.1.2 ICT industry members

While academics have a good understanding of the careers relevant to their graduates, it is also important to get relevant industry members to identify the ICT graduate roles. Nagarajan and Edwards (2008) encouraged academics to collaborate “*with industry so as to incorporate the elements that are crucial for employability of graduates as a part of curriculum development, design, training and assessment*”.

Each industry member should review each career definition and rate them on a similar scale to that used by the ICT academics. To ensure that the roles are actually available and attainable, each industry member should additionally rate each career as:

- *Employed* — have employed a (Bachelor’s level) graduate into this role in the last three years.
- *Would Employ* — would employ a graduate into this role if a vacancy existed.
- *Not Graduate* — would not employ a graduate into this role.
- *Not Relevant* — not relevant to my organisation.

Our Resource — Industry members

Three industry forums were held and eighteen representatives of the local and national ICT industry and Government participated in an exercise to identify career outcomes. The types of organisations represented were: IT recruitment, IT service/consulting, information management, IT security, research, software development, hospitality, tourism, gaming, transport, retail, fishing aquaculture, food processing, engineering, education, and government (federal and local)

Our attendees represented organisations with varying number of ICT employees from one to thousands. Nearly all had employed graduates into various positions throughout their career; most less than ten, some as high as fifty or more.

Our Output — Industry members

Only a small number of roles, 5 (out of 55), received 10 (out of 18) or more *Fully* votes from the industry representatives. There were 12 roles that received over 75% (14 out of 18) of the votes when combining the *Fully* and *Partial* votes. The results are shown in Table 2. The careers that are different from the ICT academic responses are shown in italics. The number in brackets indicates the number of attendees that indicated they have or would employ a graduate into the role.

>50% Fully votes	Fully + Partial > 75%
Data Modeller	Systems Analyst
Software Designer	Help Desk Operator
Software Developer	Network Analyst
Web Developer	Security Specialist
Database Administrator	Business Process Modeller
<i>Systems Administrator</i>	Project Support Officer
<i>Project Manager</i>	Multimedia Designer
<i>Games Developer</i>	Multimedia Developer
	Technical Architect
	<i>Security Architect</i>
	<i>Testing Manager</i>
	<i>Network Manager</i>
	<i>Information Management Specialist</i>
	<i>Solutions Architect</i>
	<i>Technical Development Manager</i>
	<i>ICT Manager</i>

Table 1: Roles identified by working party

2.1.3 Review potential roles

On completion of the initial selection activity, it is necessary to have a reflective discussion with industry members to share and discuss any differences in outcome identification amongst the industry members and with the careers identified by the ICT academics. The discussion should also consider the impact of any constraints.

Our Output — Review potential roles

The most interesting and relevant points from a discussion between all parties who had participated in the career outcomes exercise were:

- employers commonly place graduates in a Help Desk Operator role initially to test competence, and if they show ability, they are quickly advanced to a Systems Administrator or Software Developer role;
- industry members believed the role of Graphics Designer was attainable and of high demand, however it was questioned whether this role was likely to be attained by graduates solely undertaking an ICT degree, as specific skills would be required from Fine Arts related units;
- only two industry members identified the Game Developer role as one that should be *Fully* achieved, but all recognised that this role was a strong draw card for students and they welcomed the potential increase in graduate numbers it provides; and
- industry members believed it was essential that graduates were exposed to concepts in project management and business analysis during their degree but that a graduate could not enter into a Project Manager or Business Analyst role without job experience. Once shown competent, they would be rapidly promoted to these roles.

>50% Fully votes	Fully + Partial > 75%
Data Modeller (14)	Systems Analyst (12)
Software Designer (12)	Help Desk Operator (11)
Software Developer (16)	Network Analyst (7)
Web Developer (12)	Security Specialist (10)
Database Administrator (11)	Business Process Modeller (11)
	Project Support Officer (10)
	Multimedia Designer (5)
	Multimedia Developer (7)
	Technical Architect (8)
	<i>Graphics Designer</i> (9)
	<i>Business Analyst</i> (11)
	<i>Project Manager</i> (10)

Table 2: Roles identified by industry members

Even the best externally validated list of career definitions may be missing some key roles that are particular relevant to an institutions particular circumstances. For example, there may be a significant local industry sector or a significant key research/innovation direction for the university or state. A high-quality and focused degree will also potentially attract students into research.

Our Output — Missing roles

The ICT careers stream diagram was very focused on business careers and does not include titles that might fall under ICT Scientist or ICT Researcher. These careers are not necessarily of high interest to industry, but they are of significant interest to the University and the School and other research institutes within Tasmania particularly with the introduction of the NBN, Sensing Tasmania, CSIRO ICT Centre and the HITLab. As a consequence ICT Researcher was added to the list of potential roles.

2.2 Stage 2: Identifying potential SFIA skills

Stage 1 of our process identified a list of potential roles deemed desirable to use as a guide for the new curriculum development. While this list is a useful starting point, the next stage is to determine the specific skills required for the attainment of these roles by graduates.

The Skills Framework for the Information Age is owned by The SFIA Foundation (SFIA 2012). The SFIA provides a common reference model for the identification of the skills needed to develop effective information systems making use of ICT. SFIA provides a standardised view of a wide range of professional skills needed by people working in information technology. Specifically, it lists 96 professional ICT skills, with each skill being mapped across seven levels of responsibility.

2.2.1 Tabulate skills for potential roles

A role relies on a combination of skill development. Identifying an externally validated set of skills for the career definitions is an essential part of the process. Figure 3 shows an example of the SFIA skill set for a software developer, as identified by the QLD Government (2012).

Category	Skill/description	Level	Code
Strategy and architecture	Software development process improvement: Develops and maintains a detailed knowledge of software process improvement. Contributes effectively to identifying new areas of software process improvement within the organisation. Carries out software process improvement assignments, justified by measurable business benefits.	5 - ensure, advise	SPIM
Solution development and implementation	Programming/software development: Designs, codes, tests, corrects and documents large and/or complex programs and program modifications from supplied specifications using agreed standards and tools, to achieve a well engineered result. Takes part in reviews of own work and leads reviews of colleagues' work.	4 - enable	PROG
Solution development and implementation	Testing: Accepts responsibility for creation of test cases using own in-depth technical analysis of both functional and non-functional specifications (such as reliability, efficiency, usability, maintainability and portability). Creates traceability records, from test cases back to requirements. Produces test scripts, materials and regression test packs to test new and amended software or services. Specifies requirements for environment, data, resources and tools. Interprets, executes and documents complex test scripts using agreed methods and standards. Records and analyses actions and results, and maintains a defect register. Reviews test results and modifies tests if necessary. Provides reports on progress, anomalies, risks and issues associated with the overall project. Reports on system quality and collects metrics on test cases. Provides specialist advice to support others.	4 - enable	TEST
Strategy and architecture	Technical specialism: Maintains an in-depth knowledge of specific technical specialisms, and provides expert advice regarding their application. Can supervise specialist technical consultancy. The specialism can be any aspect of information or communication technology, technique, method, product or application area.	5 -ensure, advise	TECH

Figure 3: Software Developer Career and SFIA skills, included with permission of QLD Government (2012) and text from SFIA (2012) quoted by kind permission of The SFIA Foundation

To identify a potential list of skills for a degree, each role identified as a potential role should be considered and the required skill set tabulated. The process will identify some SFIA skills to include that are necessary for a number of roles and a number of SFIA skills that are not needed for any potential role.

Our Output — SFIA skills
 The subset of SFIA skills needed for our potential careers is shown in Table 3. The code titles can be found at SFIA (2012). There were a number of skills that were necessary for a range of career outcomes e.g. CNSL (Consultancy) was required for 16 careers; EMRG (Emerging technology monitoring) was required for 8 careers. We identified only 37 potential skills out of the 96 defined by SFIA.

2.2.2 Review potential SFIA skills

SFIA has identified 96 skills and based on the career outcomes identified a subset of these will be developed in a degree. A review of all the SFIA skills should be conducted to see if any essential skills have been missed. It maybe that the set of career definitions and the skill mappings did not cover the full list of SFIA skills. Consideration should also be given to any constraints during this review.

Our Output — Review potential SFIA skills
 When reviewing all the SFIA skills that were not included, we discovered two that we will consider including: HFIN (Human factors integration) and UNAN (Non-functional needs analysis) both to level 5. Both these skills relate to the recommendation in the ACM IT curriculum (ACM 2012) that user-centeredness become a pervasive theme. An analysis of all the skills required for all careers in the QLD ICT careers stream diagram identified that these skills were not listed as a specific skill for any career.

2.2.3 Feedback to role list

A significant part of the adaption of the ACS process is the feedback that occurs in each stage that influences the list of potential career outcomes. Once the list of potential skills is identified, these skills can be used to influence the list of potential roles.

There will be some skills that are only needed for a few roles and, if there are constraints, consideration can be given to removing these roles or not developing those specific skills and only partially qualifying a graduate for these roles.

There might be some roles that were not in the list of potential roles but an analysis of the skills required might identify that they are all being covered and the role could be included in the list of potential roles.

Our Output — Feedback to role list
 There were some roles identified by academics that were not identified by the industry members that require a specialist skill that within our limited number of units cannot be included such as Network Manager, Solutions Architect, or Information Management Specialist. Graduates will be partially qualified for these roles.
 There were some roles that required specialist skills, e.g. Multimedia Designer, Multimedia Developer and Graphics Designer. It was determined again within the limited number of units not to focus in this direction within this degree.
 There were some roles that were not identified as potential roles for which all the skills will be covered as a result of the careers chosen, e.g. Benefits Analyst, Animator, Hardware Engineer, Customer Services Manager, Incident Manager, and Change Manager.

2.3 Stage 3: Identify level of responsibility

To this point, a list of potential roles and the specific set of skills required for the attainment of each role have been identified. The third stage of the process is to identify the level of responsibility for each skill required to perform each role. Level of responsibility refers to the recognition that people exercise skills at different levels. SFIA recognises seven levels of responsibility ranging from 1 at basic entry to 7 at a very senior level, normally in a large organisation. SFIA's levels are: 1. Follow; 2. Assist; 3. Apply; 4. Enable; 5. Ensure/advise; 6. Initiate/influence; and 7. Set strategy/inspire/mobilise (SFIA 2012).

2.3.1 Tabulate level of responsibility

All the identified skills should be reviewed against the SFIA levels of responsibility to determine the extent to which the skill could be developed in an undergraduate degree. The process will identify skills across a range of levels. This process is guided by the ACS who recommends that undergraduate degrees should be producing graduates with skills around SFIA level 4 (enable) of responsibility (ACS 2011).

There might be a number of roles that required skills at level 5. Decisions need to be made about which level 5 skills to aim for by creating a depth of skill development throughout all years of the degree. Some of the final development might be achieved in the first six months of employment.

There might be some skills at level 5 that will not be achievable within the constraints and will only be developed to level 4. There might be some skills that have a lowest level of 5 that are not possible to achieve at university undergraduate level and might take one to two years of employment to attain.

Our Output — Level of responsibility

The levels of responsibility for all skills are also shown in Table 3. Given the constraint of 30 units the TEST (Systems Testing), ICPM (Information Content Publishing), and CHMG (Change management) skills were reduced to level 4 as we will be unable to develop these to level 5 by including units at all years of the degree. CIPM (Change Implementation Planning and Management), BENM (Benefits Management), and RLMT (Stakeholder Relationship Management) have a lowest level 5 but these skills require experience to fully achieve based on feedback from industry members.

2.3.2 Feedback to role list

There might be some roles that were selected that have skill levels that are too low or too high for an undergraduate degree. Removing these careers from the list can be considered.

Our Output — Feedback to role list

The role Help Desk operator requires SLMO (Service Level Management), ITOP (IT Operations), and USUP (Service Desk and Incident Management) at responsibility level 2. Help Desk Operator was eventually considered a side-effect career outcome of the degree, and was not seen as a role worthy as a career outcome for a university degree (people can do this with certificate IV or V). Many students take on these roles before graduation.

Project Support Officer required PROF (Project Office) at responsibility level 3 and Data Modeller required DTAN (Data Analysis) at responsibility level 3 but both also required skills at level 4; these were seen as good graduate roles that students could use to enter the workforce.

Technical Development Manager required CNSL at level 6 which can only be developed to at most level 5 in an undergraduate degree and ICT Manager required ITMG (IT Management) at level 6 — this skill was not required for any other career. As both careers also require other skills that we will develop, it was decided these would be partially qualified roles for the new degree.

2.4 Stage 4: Identify final career outcomes

Using combined insight developed from the previous three stages — identification of potential roles, identification of required skills for each potential role, and identification of each level of responsibility required for each skill — an informed decision can be made about the final set of career outcomes that would be attainable for the students, and would therefore guide the curriculum development into the future.

Given that not all potential roles and skills identified will be deemed attainable by undergraduate students immediately on completion of their studies, it is necessary to develop a categorisation that distinguishes the differences in the attainability of these career outcomes. Thus, the career outcomes are divided into four categories:

- graduate roles: all skills would be fully developed and the role is suitable for graduates (though they may need six months of experience to reach the specific level of responsibility);
- career roles: all theoretical skills would be covered and the role is suitable for graduates who have acquired one to two years of experience and shown competence;
- partially qualified roles: some key skills may be absent from the undergraduate degree which might be available from another discipline of the university or other educational institution or in a postgraduate degree; and
- non-goal roles: all the skills would be developed however the delivery of the unit content and discussion would not be focused towards these particular roles.

Graduate Roles (entry roles)
Data Modeller
Business Process Modeller
Systems Analyst
Project Support Officer
Software Designer
Software Developer
Web Developer
Games Developer
ICT Researcher/Scientist
Career Roles (after 1 or 2 years experience)
Project Manager
Business Analyst
Systems Administrator
Database Administrator
Security Specialist
Security Architect
Technical Architect
Network Analyst
Non-goal Roles
Benefits Analyst
Animator
Hardware Engineer
Customer Services Manager
Incident Manager
Partially-qualified Roles
Testing Manager (missing TEST level 5)
Help Desk Operator (missing USUP, SLMO)
Network Manager (missing NTDS, ITMG)
Information Management Specialist (missing IRMG)
Graphics Designer (missing INCA)
Multimedia Designer (missing INCA)
Multimedia Developer (missing INCA)
Change Manager (missing CHMG level 5)
Solutions Architect (missing ARCH)
Technical Development Manager (missing level 6)
ICT Manager (missing level 6)

Table 4: Final list of career outcomes

3 Conclusions: What others can learn from our process

Curriculum design is a complex process that must be informed by stakeholders and developed from multiple perspectives. In creating a new ICT curriculum we determined a need to identify those careers that would be attainable by our graduate students and guide our future curriculum design process. While career outcomes seem a logical place to commence curriculum design, there exists little direction available to guide the process of identification and evaluation of potential career outcomes and the required skills for each.

This paper reports on the development of an ICT curriculum that was guided by the ACS recommended process for developing curricula and provides practical suggestions for undertaking the first three steps:

- Have a range of academics and industry members select the roles within the constraints using an externally validated set of roles with clear definitions.

Our Output — Final career outcomes

In total we have identified 33 career outcomes for our degree. The categorisation resulted in the identification of 9 graduate roles that would be immediately attainable by our graduating students and would thus be our primary focus in developing an ICT curriculum. In addition to these core ICT graduate roles, we also identified 8 career roles, 11 partially qualified roles; and 5 non-goal roles. A list of these careers is presented in Table 4.

We will develop 31 skills (twelve to level 4, sixteen to level 5, and three to almost level 5 but experience is needed to achieve that level of responsibility). The skills will be embedded throughout the units, and each unit will work towards developing a number of skills.

Having identified the career outcomes, skills and level of responsibility we went onto complete steps 4, 5, and 6 of the ACS process (ACS 2011) as documented in Herbert et al (2013). The next step for us is to create the units based on external curricula; in particular we are using the ACM international curricula (ACM 2012). Having completed the first draft of the framework we have identified the equivalent of 28 units to cover these skills at the required level. The units will be developed throughout 2013 for delivery in 2014.

- Using the roles selected, identify the skills relevant to each career. Then use the skills identified to modify the list of potential roles (to both remove some options and introduce others) within the constraints.
- Identify the level of responsibility for each skill within any constraints. Use the level of responsibility for each skill to modify the list of roles and to classify roles into graduate roles, career roles and partially qualified roles that can be used for accurate marketing of the degree.

In following the stages of this process, career outcomes can be identified that are informed by a balanced view of academic insight and employer needs, both being further supported by externally validated and industry-standard skill definitions. Furthermore potential students can be assured that the career outcomes as stated in marketing materials are really attainable, and that the degree was developed with these career outcomes in mind.

Most readers will be interested in our reflection of using career outcomes as the mechanism for identifying the skills to be developed during a degree. Following the process documented in this paper has succeeded in producing an ICT degree curriculum and given the participants confidence that by following this process a curriculum development team can:

- determine exactly what career outcomes from the degree will be covered completely, which will be covered partially, and which will not be covered at all;

- be guided by career outcomes when developing and making decisions about what skill set to include in specific curricula;
- avoid the problems of outspoken individuals having undue influence on curricula; and
- reduce the number of units to operate within budgetary constraints, allowing time for staff to do research, and still offer a broad range of career outcomes to meet student and industry demand.

4 References

- ACM, Association for Computing Machinery (2008), *Information Technology 2008: Curriculum Guidelines for Undergraduate Degree Programs in Information Technology*.
<http://www.acm.org/education/curricula/IT2008%20Curriculum.pdf>, Accessed 26 Oct 2012
- Australian Computer Society (2011), Australian Computer Society (2011): Accreditation Manual, ACS.
- Alexander, P.M., Holmner, M., Lotriet, H. H., Matthee, M. C., Pieterse, H.V., Naidoo, S., Twinomurizi, H. and Jordaan, D., (2010), Factors Affecting Career Choice: Comparison between Students from computer and other disciplines, *Journal of Science Education and Technology*. Springer, 16 October 2010.
- Babin, R., Grant, K. and Sawal, L., (2010), Identifying Influencers in High School Student ICT Career Choice. *Information Systems Educational Journal*, 8(26).
- Biggers, M., Brauer, A. and Yilmaz, T., (2008). Student Perceptions of Computer Science: A Retention Study Comparing Graduating Seniors vs. CS Leavers. *ACM SIGCSE'08*, 12–15 March 2008, Portland Oregon, USA, 402–406.
- Calitz, A.P., Greyling, J.H., Cullen, M.D.M., (2011), ICT Career Track Awareness amongst ICT Graduates, *ACM SAISSIT'11*, October 3–5, 2011, Cape Town, South Africa, 59–66.
- Elizondo-Montemayor, L., Hernandez-Escobar, C., Ayala-Aguirre, F., & Aguilar, G. M. (2008). Building a sense of ownership to facilitate change: The new curriculum. *International Journal of Leadership in Education*, 11(1), 83–102.
- Gruba, P., Moffat, A., Søndergaard, H., & Zobel, J. (2004), “What Drives Curriculum Change?” in *Proceedings of the Sixth Australasian Computing Education Conference*, pp 109–117, ACS.
- Henkel, M. and Kogan, M. (1999), Changes in curriculum and institutional structures, in C. Gellert, ed., *Innovation and Adaption in Higher Education*, Jessica Kingsley Publ., 116 Pentonville Road, London, N19JB, England, Chapter 2.
- Herbert, N., Dermoudy, J., Ellis, L., Cameron-Jones, M., Chinthammit, W., Lewis, I., de Salas, K., Springer, M., (2013), Industry-Led Curriculum Redesign, To appear in *Proceedings of the Fifteenth Australasian Computing Education Conference*, ACS.
- Monash University 2012 Handbook, <http://monash.edu/pubs/2012handbooks/units/index-byfaculty-it.html>, Accessed 8 August 2012.
- Nagarajan, S. & Edwards, J. (2008), “Towards Understanding the Non-technical Work Experiences of Recent Australian Information Technology Graduates” in *Proceedings of the Tenth Australasian Computing Education Conference*, pp 103–112, ACS.
- QLD Government, Chief Information Office, Department of Science, Information Technology, Innovation and the Arts,
<http://www.qgcio.qld.gov.au/qgcio/projectsandservices/ictworkforcecapability/Pages/ICTcareerstreams.aspx>, Accessed 8 August 2012.
- QUT, Queensland University of Technology 2012 Handbook,
<http://www.qut.edu.au/study/courses/bachelor-of-information-technology>, Accessed 8 August 2012.
- SFIA Foundation, Skills Framework for the Information Age. <http://www.sfia.org.uk>, Accessed 8 August 2012.
- University of Swinburne 2012 Handbook,
<http://courses.swinburne.edu.au/courses/Bachelor-of-Information-Technology-I050/local>, Accessed 8 August 2012.
- UTAS, University of Tasmania 2012 Handbook,
http://courses.utas.edu.au/portal/page/portal/COURSE_UNIT/UTAS_CU_ENTRY?P_CONTEXT=NEW, Accessed 8 August 2012.
- von Kinsky, B. (2008), “Defining the ICT Profession: A Partnership of Stakeholders.” in S. Mann & M. Lopez (Eds.), *Proceedings of the 21st Annual NACCQ Conference* (pp. 15–21). Auckland, New Zealand: NACCQ.

Student Concerns in Introductory Programming Courses

Angela Carbone

Monash University
angela.carbone@monash.edu

Jason Ceddia

Monash University
jason.ceddia@monash.edu

Simon

University of Newcastle
simon@newcastle.edu.au

Daryl D'Souza

RMIT University
daryl.dsouza@rmit.edu.au

Raina Mason

Southern Cross University
raina.mason@scu.edu.au

Abstract

Student evaluations of courses across many Australian universities typically give students the option to comment on the best aspects of a course and those aspects that they believe need improving. Such comments have been collated from students in introductory programming courses at four Australian universities. In this paper we present the results of a thematic analysis to see whether there are common themes to the areas students consider most in need of improvement. We have undertaken this study to gain an understanding of the student concerns in introductory programming courses, in the expectation that a framework could be developed to assist academics with reviewing their courses in subsequent offerings. We have found that at all institutions the main focuses of student comments are the course as a whole and the assessment, although at different universities the comments focus on different aspects of these items.

Keywords: ICT Education, education quality in ICT, teaching strategy, thematic analysis

1 Introduction

Universities in Australia and elsewhere are increasingly being called to account for the quality of their student experience, which necessarily includes the perceived quality of their courses and of their teaching. Student evaluations of courses and teaching have long been standard practice in most Australian universities (Ramsden 2003). To these can be added government-administered surveys addressing similar questions (Australian Graduate Survey 2012). Now, more than ever, the findings from these surveys are being used to identify how courses and degrees can be improved.

Student surveys are usually administered towards the end of each semester, and results are analysed to provide a snapshot of students' perceptions of their teachers, the course, and their learning. In most of these

surveys, students rate a number of aspects of the course on a Likert scale, followed by open-ended questions in which they can identify best aspects of the course and the teaching, along with areas that could be improved.

Brookfield (1995) suggests that courses should be viewed through several lenses in order to improve them. Carbone and Ceddia (2012) have used student survey responses as one of those lenses, analysing the responses to develop a picture of the areas of dissatisfaction expressed by students about their ICT courses. This paper extends the work of Carbone and Ceddia, offering more focus in one respect and more breadth in another.

While Carbone and Ceddia dealt with ICT courses broadly, we have analysed student qualitative comments only from introductory programming courses. The focus on programming is appropriate because for many years authors have expressed concern about high failure rates and high attrition specifically in introductory programming courses (Denning & McGettrick, 2005; Moura, 2009; Kinnunen and Malmi 2006; Beaubouef and Mason 2005; Chalk 2003).

Second, the analysis is undertaken across four universities: two major metropolitan universities, a large non-metropolitan university, and a smaller regional university with a significant distance education component, to enhance the generalisability of our findings. This spread enables us to explore which student concerns are common across multiple universities.

Much work has been devoted to ways of teaching that might improve the students' learning, the students' learning experience, or both; but little work has been reported on the students' own experiences of the course and how it can be improved. In this paper, we have set out to explore what aspects of introductory programming courses are seen by students as most in need of improvement. We have two specific research questions:

1. *What do students perceive as the major concerns in introductory programming courses?*
2. *Are the students' concerns common across different institutions?*

This paper reports on a thematic analysis of the qualitative comments from student evaluations in which students suggest areas for improvement in introductory programming courses from four universities.

Copyright © 2013, Australian Computer Society, Inc. This paper appeared at the 15th Australasian Computer Education Conference (ACE 2013), Adelaide, South Australia, January-February 2013. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 136. A. Carbone and J. Whalley, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

2 Background

Student evaluation surveys focus on student experiences of courses, and are extremely important in identifying whether courses are meeting students' expectations and needs, as well as areas that can be improved. However, as Galbraith et al (2012) note, survey evaluations are in no way a measure of student learning. It is possible for students to like a course and learn very little from it, or vice versa. Also, as Pears (2010) suggests, the student is not necessarily the most competent stakeholder in the education process to gauge course quality. Even so, the students' experiences of the course are clearly important. Lefevre (2012) has shown that lecturers who were specifically asked to alter their presentation content in light of student feedback showed a greater than average one-year change in student evaluation scores. This finding confirms that courses can be enriched by changes based upon student feedback.

2.1 The Student Evaluation Instruments

The student evaluation surveys used at the four participating institutions are described below, with particular focus on the free-text questions whose answers we have analysed.

For items requiring a closed response, all instruments use a five-point Likert scale ranging from *Strongly disagree* (1) to *Strongly agree* (5), with 3 representing *Neutral*. Options for *Not applicable* and *Don't know* were also provided in some cases; these options are not counted when calculating the mean responses for questions.

Reports generated from the analysis of the closed question responses for all courses are generally accessible by staff and students of the university in question. Access to the qualitative comments is restricted to academic staff and their supervisors. Usually the survey is issued in the last 4 weeks of the session, and is available to all students enrolled in that course, irrespective of location or study mode.

Monash University

The instrument used at Monash University is distributed online at the end of each semester. There are five university-wide Likert-scale course evaluation items:

1. *The unit enabled me to achieve its learning objectives.*
2. *I found the unit to be intellectually stimulating.*
3. *The learning resources in this unit supported my studies.*
4. *The feedback I received in this unit was helpful.*
5. *Overall I was satisfied with the quality of this unit.*

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?*

For the analysis in this study we collected the answers to the second of these questions.

RMIT University

The Course Experience Questionnaire (CEQ) at RMIT University has 13 Likert-scale questions of the form:

The learning objectives of this course are clear to me (Strongly agree) to (Strongly disagree).

There are also two open-ended questions:

1. *What are the best aspects of this course?*
2. *What aspects of this course are most in need of improvement?*

For the analysis in this study we collected the answers to the second of these questions.

The University of Newcastle

The survey instrument used at The University of Newcastle has 15 two-part questions and two further open-ended questions. The first part of each two-part question is closed, and the second part is a free-text question inviting further comment on the same topic. For example,

Q12. FEEDBACK: I received feedback that was helpful to my learning.

Q12a. Any comments regarding feedback?

The additional open-ended questions are:

Q16. Are there things about this course that you have not already mentioned that you think are particularly good?

Q17. Are there things about this course that you have not already mentioned that you think need improving?

For the analysis in this study we examined all of the free-text responses to any of the 17 questions and collected those that suggest some need for improvement in the course.

Southern Cross University

Southern Cross University has 18 Likert-scale questions, of which seven are system-wide course-related questions, five are questions selected by the course instructor, and six are system-wide teaching-related questions.

Each of these questions has an associated open question: "please explain your reasons for your rating".

There are two additional open questions:

1. *Here is your opportunity to tell us how to improve this unit.*
2. *Here is your opportunity to give other feedback on the teaching in this unit.*

For the analysis in this study we collected all comments that could be considered to be criticisms or suggestions for improvement.

3 Research Context

The four institutions participating in this study are demographically quite varied; a brief description of the background of each course at the institutions is presented below. Note: what is called a *unit* in some Australian universities is often called a *course*, a *subject*, or (in New Zealand) a *paper*. The term 'course' will generally be used in the remainder of this paper.

3.1 Monash Context

Monash University offers Introductory Programming as a core course to four undergraduate degrees. The course is offered across multiple campuses: four domestic

campuses at locations within Australia and two international campuses. This introductory programming course is offered at all six campuses.

The assessment consists of a 3-hour examination worth 60% and in-semester assessment worth 40%. The in-semester assessment is broken into three assignments worth 5%, 10% and 15%, and a mid-semester test worth 10%, with ViLLE quizzes used as a hurdle. ViLLE (ville.utu.fi) is a programming visualisation tool that offers a range of question formats for Java and other languages. At Monash, ViLLE is employed principally for non-assessed work in Introductory Programming.

3.2 RMIT Context

Programming 1 is a first-year course that teaches foundations of programming with Java as the vehicle of instruction. It is a core course for students enrolled in the Bachelor of Computer Science, Bachelor of Software Engineering, and Bachelor of IT. For the latter course students must have completed Introduction to Programming or have equivalent recognised prior learning. Programming 1 is also offered to postgraduate coursework students whose first degree is not in Computer Science or cognate disciplines.

ViLLE was employed for both assessed and non-assessed work.

Assessed work comprises 3 assignments (total 35%), regular tutorial quizzes (10%), a mid-semester paper test (4%), a mid-semester ViLLE online test (6%), an online exercise (5%), and a final examination (40%). The final exam and the other assessed work form two separate hurdles.

3.3 Newcastle Context

Visual Programming is an introductory programming course that is a core requirement for students in the Information Technology degree, although it is also taken by a number of students in other degrees. At the time of this study it was taught using C# in Visual Studio.

The assessment in Visual Programming consists of two practical tests each worth 15%, a paired assignment worth 20%, and a final written exam worth 50%.

3.4 Southern Cross Context

We have examined two first-year introductory programming courses from Southern Cross. The first uses Visual Basic .NET as its programming language and has four minor assignments (small programs), one major assignment (larger program) and an exam. The second course uses C# as its programming language and has three online quizzes (based on previous programming exercises), one minor assignment, one major assignment, and an exam.

Both of these courses have lectures that are delivered live to all students once a week via Blackboard's Collaborate software (<http://www.blackboard.com/platforms/collaborate/overview.aspx>). Recordings of the lecture are also made available for those students who are not able to attend the online lecture, via the university's Blackboard site for each course. In addition, in each course, six live online workshops are

conducted via the Collaborate software, which are also recorded for students who cannot attend. On-campus students attend internal workshops of two hours/week. Both on-campus and external students are expected to work through a provided study guide.

For the purposes of this analysis we consider these two courses to be a single introductory programming course.

3.5 Overall Course Satisfaction

While this research focuses on the students' free responses to the open-ended questions, there is one Likert-scale question that is often used to benchmark courses, and that is the overall satisfaction question, which is something along the lines of:

Overall I was satisfied with the quality of this course.

This question is typically used as a quality measure to rank courses and to identify those that are most in need of attention. At Monash, for example, courses with a mean response of 3 or less for this question are flagged as needing critical attention, and courses with a mean response from there up to 3.6 are flagged as needing to improve. The dividing lines are not the same at each university, and are determined at different levels of the university hierarchy, but the principle remains more or less the same.

Table 1 lists, for each institution, the enrolment in the course we were examining, the survey response count and rate, and the score for the course on the overall satisfaction question. None of the programming courses that we have studied fared particularly poorly on this question.

Where the overall satisfaction score appears as a range, it is because the responses were gathered from two or more offerings of the course, each of which had a different score.

3.6 Response Rates and Comment Counts

The response counts listed in Table 1 are the formal counts of the numbers of students who completed the surveys. There is no clear relationship between these response counts and the numbers of open-ended answers received. A student might respond to some or all of the closed questions and none of the open questions. For example, at RMIT there were 211 enrolments with 80 responses, giving a response rate of 38%. However, there were only 46 actual qualitative comments, giving a 'comment' response rate of 22%. For Monash the comment response rate is 27%, and for Newcastle it is 22%.

A further complication is that while Monash and RMIT each give students a single opportunity to suggest aspects of the course that might need

Institution	Enrolment	Response count	Response rate	Overall satisfaction
Monash	674	234	35%	3.63-4.33
RMIT	211	80	38%	3.75
Newcastle	301	92	31%	3.83-4.35
Sthn Cross	236	71	30%	3.77-4.00

Table 1: Course demographics

improvement, Newcastle and Southern Cross give multiple opportunities for open comment. When the comments are made available in summary form there is no way to discern whether two or more of them were made by the same student. It is possible, for example, for a single student to have made a dozen or more comments on the same form. Indeed, there is evidence that some students did contribute two or more comments, because some comments begin with words such as ‘As I remarked above . . .’

For these reasons the comment counts are best considered simply as comment counts, with no useful connection to the survey response rates.

4 Research Approach

This section describes the processes of data collection and analysis.

4.1 Data Collection

Reports on the quantitative course evaluation results are generated by central university areas, and are publicly accessible by staff and students by way of the university websites. However, access is more controlled to the responses to the open questions.

Human ethics approval was obtained from each institution to analyse the course evaluation qualitative comments for the introductory programming courses. This approval entailed gaining consent from the staff involved in teaching the courses.

Once ethics approval had been granted and the lecturers’ consent had been obtained, the qualitative data was provided by the university statistics units at each participating university. All files with qualitative data had any identifying lecturer and tutor information removed, along with any further sensitive information that might offer the potential for re-identification.

4.2 Focus of Analysis

A thematic analysis was undertaken on the student responses to the following open-ended question or its equivalent:

What aspects of this course are most in need of improvement?

4.3 Deriving a Common Set of Categories

As a convenience, and to avoid reinventing the wheel, we began with the set of categories and subcategories derived by Carbone and Ceddia (2012). The main categories were straightforward to identify as they were often key words in the comment. For example, a comment beginning with “The lecturer was...” would normally (though not always) be classified into the *Lecturer* category.

Each researcher began by classifying all of the comments in one dataset. The team then met to discuss differences, clarify understanding of the classification scheme and of the course in question, and introduce new categories or subcategories if required. At times it was necessary for the member from the institution being classified to explain any special software or process that was used at that institution. For example, two of the universities use the ViLLE program visualisation

system (ville.utu.fi) in their programming courses, but one uses it just as a practice tool while the other also uses it for an aspect of assessment. The researchers from those universities had to explain to the others what ViLLE was and how it was used in their particular courses so that students’ comments about ViLLE could be accurately interpreted and classified.

Discussion of the data from each university resulted in a possibly revised set of categories to be used when classifying the next dataset. This was done progressively with the data from the four universities. Thus the derivation of the categories and subcategories was addressed naturally and progressively through four rounds of coding and discussion.

In the course of our analysis we developed four further categories in addition to those used by Carbone and Ceddia (2012). However, we agreed that two of these contribute nothing to the lecturer’s understanding of how to improve the course, so we have labelled them ‘non-contributing’ categories.

The first of these categories is *Null*. The student clearly intended to comment on something that needed improving in the course, but the comment itself was so uninformative that we were unable to classify it. An example of a Null comment is the single word “practice”. Such comments contribute nothing useful to the lecturer planning the delivery of a course, so we have omitted them from the summary counts and from further analysis.

The second non-contributing category is *Student*. Comments in this category came principally from Newcastle, where one question asks students if they have any comment on the effort they put into the course. While teaching staff clearly have some influence on how students approach a course, after examining the specific comments in this category we concluded that nothing they said would be of help to lecturers in preparing to deliver a course, so these, too, were removed from consideration.

The two additional contributing categories are *Course* and *Lab*. When students’ comments pertained to the course as a whole, rather than any specific aspect of it, we felt that this merited its own category. Students might, for example, suggest that improvements could be made to the course content, the overall course structure, or the relevance of the course to their other studies or to their career aspirations.

When we found comments pertaining to computer lab classes our first inclination was to classify them into the *Tutorial* category. However, the course at Monash is taught by way of lectures, tutorials (in which the students do pen-and paper exercises), and labs (in which they carry out programming at computers), so we introduced the *Lab* category in recognition of this distinction – while recognising that at other universities there might be some blurring of these two categories.

In addition to the four new categories we discerned several new subcategories. One important subcategory is *unspecified*. It is not uncommon for a student to clearly indicate what needs improving, but not in what regard. For example, a comment that “lectures” need improving might be referring to the content, timing, duration, structure, or some other aspect of the lectures.

We put such a comment into the *Lecture* category so that it would be counted there, but for lack of further information then put it into the *unspecified* subcategory.

The final ten categories and their subcategories are listed in Appendix 1.

4.4 Inter-rater reliability

It is common to measure inter-rater reliability (Banerjee et al, 1999) when undertaking any form of subjective classification, such as, in this research, the classification of each student comment into a category and subcategory. This measurement is generally done for one of two reasons: to validate the classification system being used, or to ensure that individual classifiers are classifying comparably when the work is partitioned among them.

The purpose of this study was to determine common concerns in introductory programming courses, and not to propose a classification system for others to use, so there was no need to validate the system.

Furthermore, the task of classification was not partitioned among the researchers, so it was not necessary to establish that each classifies as the others would; the researchers classified all the data separately, but then met to discuss and reach consensus on a classification for each and every comment. For these reasons we did not carry out inter-rater reliability calculations.

5 Results

A total of 347 qualitative comments from four introductory programming courses were categorised and coded. The actual number of individual students who gave feedback is less than 347, as some students commented on multiple areas and these were coded as separate comments into their respective categories. In particular, at Monash and RMIT there was a single question asking students to nominate aspects of the course that could be improved; some students combined multiple suggestions into these single comments, and when classifying the comments we separated them into their component parts and classified each separately. Thus a single comment might be counted and classified as several. This gives some comparability with Newcastle and Southern Cross, where the surveys give multiple opportunities to comment, each on specified aspects of the course.

5.1 Comment frequencies

Tables 2, 3, 4 and 5 list the frequencies of comments in each category and subcategory for Monash, RMIT, Newcastle, and Southern Cross respectively. In these tables, the ‘%’ column is calculated as a percentage of the total comments for the institution. For example, in Table 2 for the Monash data, 51 of the 182 comments were classified in the *Course* category, giving 28%. Within each category the most prevalent subcategory is marked in bold text.

In the following subsections we shall note any classifications that appeared in some way localised to the university in question.

5.2 Concerns of students at Monash

A total of 182 comments were coded for Monash; the classifications are listed in table 2. This is the university

Category	Frequency	%	Subcategory	Freq
Course	51	28%	challenge	12
			content	10
			organisation	20
			relevance	1
			workload	5
			unspecified	3
Lecturer	16	9%	control	1
			knowledge	1
			presentation	8
			support	5
			unspecified	1
Lecture	15	8%	access	1
			content	7
			duration	2
			quality	1
			structure	2
			unspecified	2
Tutor	21	12%	organisation	1
			presentation	7
			support	13
Tutorial	23	13%	alignment	2
			clarity	2
			length	7
			scheduling	2
			type of activity	9
			unspecified	1
Lab	11	6%	activity	4
			length	3
			unspecified	4
Assessment	32	18%	alignment	3
			content	3
			difficulty	3
			feedback	1
			marking	2
			organisation	9
			practice	3
			specification	5
			timing	1
			unspecified	2
Resources	11	6%	availability	1
			quality	8
			quantity	1
			unspecified	1
LMS	1	1%	ease of use	1
OffCampus	1	1%	support	1

Table 2: Monash data - 182 comments total

that had distinct labs and tutorials, and for which we introduced the *Lab* category.

Lab – activity refers to the type of activity issued in the lab. A sample comment is:

The lab sessions weren't very productive, I was basically sitting at my computer for 2 hours working on question 1 of each task and the supervisor did nothing unless you asked him

Lab – length refers to the time allocated to the lab. A sample comment is:

computer lab time is too short to do 4 exercises.

5.3 Concerns of students at RMIT

A total of 47 comments were coded for RMIT; the classifications are listed in table 3. The data from this university led to significant discussion on the nature and role of ViLLE, so that we could be more confident whether at this university we should classify comments about ViLLE in the *Assessment* or *Resources* category.

5.4 Concerns of students at Newcastle

A total of 63 comments were coded for Newcastle; the classifications are listed in table 4. Coding of the

comments for this university led to a new subcategory, *challenge*, which emerged under the categories *Course* and *Lecture*. We would not normally expect a student to comment on the challenge posed by a course (as distinct from, say, the workload of the course). However this is one of the universities that invite comment on specific topics, and after asking students to rate the challenge of the course its survey explicitly asks if they have any further comment on challenge.

Course – challenge refers to the level of difficulty of the course as experienced by the student, with comments such as:

[Challenge] As this is my first programming course but second attempt yeah it was a challenge .

Lecture – challenge refers to the level of difficulty of the lecture material, and arises from the comment:

[Learning activities] I find this is a hard course to teach through lectures

While other comments in this paper are reproduced verbatim, the notation above indicates that a student

Category	Frequency	%	Subcategory	Freq
Course	10	21%	content	4
			organisation	1
			relevance	1
			unspecified	1
			workload	3
			Lecturer	1
Lecture	4	9%	content	1
			structure	2
			unspecified	1
Tutor	2	4%	support	1
			unspecified	1
Tutorial	2	4%	type of activity	2
Assessment	20	43%	alignment	1
			content	1
			difficulty	1
			feedback	2
			practice	1
			quantity	3
			specification	7
			support	1
			timing	2
			unspecified	1
Resources	7	15%	availability	1
			quality	5
			unspecified	1
LMS	1	2%	quality	1

Table 3: RMIT data – 47 comments total

Category	Frequency	%	Subcategory	Freq
Course	23	37%	challenge	6
			content	1
			organisation	3
			quality	1
			relevance	3
			unspecified	1
			workload	8
Lecturer	7	11%	organisation	1
			presentation	3
			support	3
Lecture	3	5%	challenge	1
			content	1
			unspecified	1
Tutor	6	10%	support	6
Tutorial	4	6%	structure	1
			scheduling	2
			type of activity	1
Assessment	12	19%	difficulty	1
			feedback	2
			marking	1
			organisation	2
			practice	1
			specification	4
			unspecified	1
			Resources	8
quality	2			
quantity	1			
readings	3			

Table 4: Newcastle data - 63 comments total

wrote “I find this is a hard course to teach through lectures” in response to the invitation “Any comments regarding learning activities?” We have used this notation as some of the comments from this university would be difficult to interpret without the context of the question to which they were responding.

5.5 Concerns of students at Southern Cross

A total of 55 comments were coded for Southern Cross; the classifications are listed in table 5. This university delivers its lectures via the Collaborate tool rather than face-to-face; some students found this interface difficult to deal with, leading to comments on delivery mode, a subcategory that we did not find at the other universities.

The subcategory *ease of study* emerged under the main category *OffCampus*. The subcategory relates to the challenges students face when undertaking off-campus courses and arose because of this comment:

Yet again, this unit posed many challenges due to the tyranny of distance and subsequent lack of hands on interaction with peers and lecturer. The blackboard is a wonderful asset for distance ed students but will never replace the vibrancy of an actual lecture or workshop. Even though [lecturer] was always open to emails the problem is that asking questions in emails can be difficult,

Category	Frequency	%	Subcategory	Freq
Course	12	22%	content	3
			organisation	9
Lecturer	8	15%	presentation	3
			support	4
			unspecified	1
Lecture	6	11%	content	2
			delivery mode	3
			structure	1
Tutor	6	11%	organisation	2
			quality	1
			response time	1
			support	2
Tutorial	7	13%	type of activity	7
Assessment	9	16%	alignment	2
			content	1
			feedback	2
			marking	2
			practice	1
			timing	1
Resources	6	11%	availability	1
			content	2
			quality	3
OffCampus	1	2%	ease of study	1

Table 5: Southern Cross data - 55 comments total

especially when you are not even sure what your asking for because the topic is so foreign.

6 Discussion

Table 6 lists top three category student concerns at each institution. The rows list the main category, the columns list the institution concerned, and the cell contents identify the subcategory for the institution. For example, the top issues for Monash are *Course – organisation*, *Assessment – organisation* and *Tutorial – type of activity*.

The two research questions that motivated this research were:-

1. *What do students perceive as the major concerns in introductory programming courses?*
2. *Are the students' concerns common across different institutions?*

These questions can now be answered by examining Table 6. The twelve completed cells indicate what students perceive as the major concerns in the introductory programming courses at these four institutions. At a category level there is fairly strong agreement: students express concerns about the course as a whole and about the assessment in the course. There is rather more diversity at the subcategory level, and the answer to the second question would seem to be that there is some commonality in the students' concerns, but that there is also diversity, and even when considering only four institutions we are able to identify concerns that relate directly to the specific circumstances of each course. These points are illustrated in the following sections.

6.1 Concerns about the course

With regard to the top category concern of *Course*, students at Monash and Southern Cross are concerned about the *course structure*, while the concern at Newcastle focuses on *workload* and that at RMIT focuses on *course content*.

Course – structure refers to the way that components of the course are arranged. Comments include:

- *The unit should emulate the structure of other subjects (e.g. Commerce subjects) where tutorials and labs cover the previous week's lectures (as opposed to the current week's lectures). [Monash]*
- *found that the subject seemed to concentrate most marks toward the end with the difficult concepts of files and structs left until the major assignment was due. Perhaps more balanced loading over the semester [Southern Cross]*

Course – workload refers to the size and number of things to do in the course. Comments include:

Just time we need more time so we can spend more time on individual things rather than cram everything into a short period of time [Newcastle]

Course – content relates to the choice of topics that are covered, including the programming language used. Comments include:

change language from java to something more widely used [RMIT]

6.2 Concerns about assessment

With regard to the top category concern of *Assessment*, students at RMIT and Newcastle are concerned about the *assessment specifications*, students at Monash about the *organisation* of assessment, and students at Southern Cross about *feedback, marking, and alignment* of the assessment with the course learning objectives.

Assessment – specification refers to the clarity with which assignments were written, the submission process, and any changes to requirements. Comments from RMIT and Newcastle include:

- *Explanation of assignments; Rehash of previous assignments properly implemented [RMIT]*
- *the rules for the game in the assignment were a little ridiculous, making the game just a little bit to near impossible to understand. [Newcastle]*

Assessment – organisation refers to the due dates and the allocation of marks to components of assessment. Comments include:

- *I don't believe the exam should have such a strong influence on your overall score as exams are not a practical programming environment. The real test of your programming skills is the second assignment. [Monash]*

Assessment – feedback refers to the usefulness of the correspondence in relation to the assessment. Comments include:

- *Feedback is also an issue, with the option for Continuous Assignments to be marked in class the feedback is severely lacking. I found many times that my documentation was looked over in 10 seconds then the code was looked over in about 30 seconds, then it was tested that everything worked, and the assignment was ticked off with very minimal feedback. When we upload our assignments to [the website] they need to be thoroughly looked over and precise feedback be given that clearly outlines what needs to be changed so that full marks can be attained. With this feedback students would be able to look through past feedback and apply it to the next assignment and get full marks easily. Students put in a lot of work to make sure assignments are done the least that can be done is that they are marked and feedback be given to the same standard. [Southern Cross]*

Assessment – marking refers to the level of consistency of marking, quality of feedback, timeliness, and clarity of marking criteria. A typical comment includes:-

- *Also for most of our Continuous Assessments, he didn't seem to care about the documentation most weeks, as long as the application worked. Then why is it apart of the work if he doesn't care? This*

makes it difficult to know exactly what it is that he wants! [Southern Cross]

Assessment – alignment refers to the degree of alignment of assessment tasks with course learning objectives. A typical comment includes:-

- *My only complaint is that the topic on open/read a file was far too late in the course. I feel this should have been delivered earlier, which would have aided in getting the major assignment well under way, earlier in the course. [Southern Cross]*

6.3 Other prevalent concerns

There is less commonality in the third category of choice, with students at Monash expressing concern about the type of activity carried out in tutorials (which, remember, are different from computer labs), students at RMIT about the quality of the resources, students at Newcastle about the readings provided as resources (which include the textbook), and students at Southern Cross about the support provided by lecturers (remembering that these courses are lectured remotely though an online collaboration system). Comments illustrative of these concerns are provided below:

Tutorial – type of activity

Tutorials - should have easier questions - takes too long to get through the tutorial worksheets (we get through it but in a bit of a rush) to enable people more time to grasp concepts. [Monash]

Resources – readings

i found that the text book was simply rubbish and its writing was ... confused at best. [Newcastle]

Resources – quality

The online tests. ViLLE was not an appropriate learning tool as I had many problems using it. [RMIT]

Lecturer – support

In the Workshops, it would be alot better if [lecturer] were to actually explain and go through pieces of code, instead of just sitting there waiting for us to ask questions. [Southern Cross]

7 Conclusion and Future Work

This study aimed to take a first step in developing an understanding of the areas that are perceived as concerns by students in introductory programming courses. This understanding was achieved by analysing the qualitative responses to the course evaluation questionnaire of first-year programming courses from four Australian universities.

The qualitative comments were obtained from each university's central units after approval from Human Ethics at each institution and receiving the lecturer's

Category	Monash	RMIT	Newcastle	Southern Cross
<i>Course</i>	Organisation	Content	Workload	Organisation
<i>Assessment</i>	Organisation	Specification	Specification	Feedback, Marking, Alignment
<i>Resources</i>		Quality	Readings	
<i>Lecturer</i>				Support
<i>Tutorial</i>	Type of activity			

Table 6: Top three category/subcategory pairs at each institution

consent. The analysis was achieved by each researcher classifying all of the comments from one university and then discussing differences, and introducing new categories if required. This process was repeated for each university's data.

A total of 347 qualitative comments were categorised into 10 main categories: Course, Lecturer, Lecture, Tutor, Tutorial, Lab, Assessment, Resources, LMS and OffCampus. Each of these main categories was divided into subcategories to provide a more fine-grained focus on the real concern for the student. The subcategories serve to define the attributes of the main category. It is these subcategories that provide the real value to lecturers so that they can interpret the students' feedback and act to achieve real improvements in their courses.

From the analysis the students across all four universities expressed common concerns about the *Course* and the *Assessment*. With regard to the *Course* the most common subcategories that emerged were related to *specification*, *workload* and *content*. With regard to *Assessment* the most common were *organisation*, *specification* and *feedback*. Other prevalent concerns varied depending on the institution.

Following the analysis we have produced a *course quality attribute* list, which is shown in Appendix 1. We believe that this output from our research will be a useful guide for academics, highlighting areas that they should consider when preparing for a new course or a new offering of an existing course. This list is not a schedule or a to-do list; rather, its purpose is to prompt academics to consider what past students have experienced as course quality, and to bear this in mind while preparing a course for delivery to students.

At this stage the list has not been deployed; that will take place during the next phase of the study. The list will be distributed to relevant lecturers of introductory programming courses at their institutions to get feedback from the lecturers themselves as to what they consider needs improvement in their courses. Each researcher will work closely with a 'buddy' academic to pilot the list and determine its applicability and usefulness. Along with deployment, the researchers will be able to validate the list and confirm its reliability.

8 Acknowledgements

We gratefully acknowledge the lecturers who were willing to let us examine the negative feedback on their courses.

9 References

- Australian Graduate Survey 2012. Graduate Careers Australia. <http://www.graduatecareers.com.au/research/surveys/australiangraduatesurvey/> [Accessed June 2012].
- Banerjee, M., Capozzoli, M., McSweeney, L., and Sinha, D. 1999. Beyond kappa: a review of interrater agreement measures. *Canadian Journal of Statistics*, 27:1, 3-23.
- Beaubouef, T. and Mason, J. 2005. Why the high attrition rates for Computer Science students: Some thoughts and observations. *ACM SIGCSE Bulletin*, vol. 37, no. 2, pp. 103—106, ACM New York, USA, June 2005.
- Bennedsen, J. and Caspersen, M. E. 2007. Failure rates in introductory programming. *ACM SIGCSE Bulletin*, vol. 39, no. 2, pp. 32—36, ACM New York, USA, June 2007.
- Brookfield, S., (1995) *Becoming a critically reflective teacher*. San-Francisco: Josey-Bass 1995
- Carbone, A., and Ceddia J. 2012. *Common Areas for Improvement in ICT Units with Critically Low Student Satisfaction*. Fourteenth Australasian Computing Education Conference (ACE 2012), Melbourne, Australia, 167-175.
- Chalk, P., Boyle, T., Pickard, P., Bradley, C., Jones, R. and Fisher, K. 2003. Improving pass rates in introductory programming. In Proceedings of 4th LTSN-ICS Conference, Galway, UK, August 2003.
- Denning, P. & McGettrick, A., 2005. Recentering Computer Science. *Communications of the ACM*, 48(11), pp.15–19.
- Galbraith, C., Merrill, G. & Kline, D. 2012. Are Student Evaluations of Teaching Effectiveness Valid for Measuring Student Learning Outcomes in Business Related Classes? A Neural Network and Bayesian Analyses. *Research in Higher Education*, 53, 353-374.
- Kinnunen, P. and Malmi, L. 2006. Why students drop out CS1 course? In Proceedings of the second international workshop on Computing education research (ICER'06), pp 97—108, ACM New York, USA, 9—10 September 2006.
- Lefevere, Kathelijne. *HERDSA 2012 conference presentation* Course evaluation: does student feedback improve future teaching? Personal communication.
- Moura, I., 2009. Teaching a CS Introductory Course: An Active Approach. In *Proceedings of Society for Information Technology & Teacher Education International Conference 2009*. Charleston, SC, USA: AACE, pp. 2308 – 2317.
- Pears, A. 2010. Does Quality Assurance Enhance the Quality of Computing Education? presented at the meeting of the 12th Australasian Computing Education Conference (ACE 2010), Brisbane, Australia.
- Ramsden, P (2003). *Learning to Teach in Higher Education*. RoutledgeFalmer, London.

Appendix 1. Course quality attributes list

These course attributes are derived from a study of student feedback comments on aspects of courses that could be improved. The attributes are divided into categories and subcategories, and within each category the highlighted subcategory is the one that drew most comments in this study.

Category	Subcategory	Description	Check
Course	challenge	The level of challenge and difficulty of the overall course	
	content	The choice of topics that are covered in the course, including programming language used	
	relevance	The real world scenarios in the course and whether the course is current	
	structure	The way that components of the course are arranged	
	workload	The size and number of things to do in the course	
Lecturer	control	The amount of control the lecturer has over disruptive students in class	
	knowledge	The amount of knowledge the lecturer portrays to the students	
	organisation	The way the lecturer arranges the components of the lecture	
	presentation	The level of engaging teaching methods used to deliver the material	
	support	The lecturer's availability and attitude towards the students	
Lecture	access	The ease with which the lecture materials can be reached by students	
	challenge	The level of difficulty of the material	
	content	The choice of topics and activities that are presented in the lecture	
	delivery mode	The suitability of the mode of delivery	
	duration	The amount of time allocated to the lecture	
	structure	The logical sequencing of concepts	
Tutor	organisation	The way the tutor arranges the components of the tutorial	
	presentation	The engaging teaching methods used to deliver the material	
	response time	How quickly the tutor responds to students' queries	
	support	The tutor's availability and attitude towards the students	
Tutorial	alignment	The alignment of tutorial activities with course learning objectives	
	clarity	The clearness of the requirements of the task	
	length	The amount of time allocated to the tutorial	
	scheduling	When the tutorial classes are scheduled	
	structure	The logical sequencing of activities	
	type of activity	The type of tutorial activity	
Lab	activity	The type of laboratory activity	
	length	The amount of time allocated to the lab	
Assessment	alignment	The alignment of assessment tasks with course learning objectives	
	content	The choice of tasks covered by the assessment items	
	difficulty	The level of difficulty of the assessment items	
	feedback	The usefulness of the correspondence in relation to the assessment	
	marking	Consistency of marking, quality of feedback, timeliness, and clarity of marking criteria	
	organisation	Due dates and the allocation of marks to components of assessment	
	practice	The amount of similar tasks students have experienced	
	quantity	The number and size of assessments	
	specification	The clarity in which assignments were written, submission process and changing of requirements	
	support	The assistance provided to students in relation to their assessment tasks	
timing	When in the teaching term the assessment items are issued and due		
Resources	availability	How accessible and ready for use a resource is	
	content	The usefulness of the resources	
	quantity	The amount of resources	
	readings	The suitability of the readings	
LMS	ease of use	The simplicity with which materials can be found on the LMS	
Off Campus	ease of study	The challenges students face when undertaking off campus courses	
	support	The assistance provided to students studying in distance education mode	

Stakeholder-Led Curriculum Redesign

Nicole Herbert, Julian Dermoudy, Leonie Ellis, Mike Cameron-Jones, Winyu Chinthammit, Ian Lewis, Kristy de Salas, Matthew Springer

School of Computing and Information Systems

University of Tasmania

Private Bag 87, Hobart 7001, Tasmania

Nicole.Herbert@utas.edu.au

Abstract

The University of Tasmania is undertaking a ‘green-fields’ replacement of its existing undergraduate ICT offerings. As part of the process over thirty industry members and educators were interviewed to gain their advice on what should be included in the only bachelors level ICT degree offered in Tasmania from 2014. This paper reports on lessons learned in ICT curriculum review and in the identification of desired graduate skills and knowledge for future employment. With a strong trend towards utilising outsourcing and off-shoring for software and system development, industry members indicated that there is no room in the ICT industry of the future for personnel who cannot relate to customers and who lack the business acumen to be able to undertake analysis at the commencement of a project or integration at its conclusion. The review identified strong demand for graduates to be ICT professionals with generic professional skills (such as communication and teamwork) along with other non-technical skills (such as business analysis, sourcing and integration) in addition to the traditional domain skills (including programming and databases). Employers desired graduates with a broad range of ICT knowledge but with a depth of competency in at least one ICT technical area. A summary of the outcomes, including likely degree content, is provided.

Keywords: ICT Curriculum, ICT Graduates, ICT Industry

1 Rationale

As has been the case in ICT schools nationally, the staff profile of the School of Computing and Information Systems at the University of Tasmania (UTAS) has contracted, by almost a third, in the last decade and this contraction is expected to continue. Further, like many Australian universities, UTAS is re-positioning itself. The University is seeking to increase its research-led reputation and to rationalize the number of units (subjects) delivered. To this end, the School of Computing and Information Systems was recently administratively reviewed and it was recommended that the current undergraduate degrees, a Bachelor of Computing and a Bachelor of Information Systems, be discontinued and a single degree be created in their place.

In addition, the Review Panel also recommended that the School reduce the number of undergraduate units from the current fifty to only thirty. In comparison to other universities, the number of units offered is already small¹.

Unlike all other Australian universities — with the notable exception of Charles Darwin University — UTAS is the only university in its state/territory, and consequently the School has to primarily meet the ICT higher education needs of Tasmania. Every one of the intended thirty units must maximize its contribution by working towards providing graduates with the essential technical and non-technical ICT skills and professional skills to enhance the Tasmanian ICT industry and/or attracting students into an ICT research career to increase the research potential of the School and/or generating as much EFTSL income for the School as possible by being attractive to non-ICT degree students.

The aim of this paper is to present the findings of our current curriculum review process, the process followed, and the lessons learned for ICT schools of other Australian universities.

2 Previous approaches

An early investigation into Australia’s ICT needs occurred in 1999 (Ignite, 1999). This identified areas of technical shortage. Domain-specific knowledge and skills, however, are not the only requirements of job-ready graduates. Nagarajan and Edwards (2008) highlighted the fast changing nature of ICT and the impact such fast-paced change has on the demands and expectations of employers. They suggested that to manage this change there needs to be close and continuous communication between universities and industry.

Despite the high rate of change, employer demands appear to have been relatively constant over time:

- The Australian Newspaper (2006) found that employers valued communication skills and people skills above academic qualifications and that applicants for positions would not be hired without well-developed communication skills.

Copyright © 2013, Australian Computer Society, Inc. This paper appeared at the Fifteenth Australasian Computing Education Conference (ACE2013), Adelaide, Australia, January 2013. Conferences in Research and Practice in Information Technology, Vol. 136. Angela Carbone and Jacqueline Whalley, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

¹ Monash has 120 undergraduate units listed (code FIT) in its 2012 handbook for the Faculty of IT (Monash 2012), Swinburne has 163 (code HET, HIT) in its 2012 handbook for the Faculty of ICT (Swinburne 2012), and QUT has 64 undergraduate units listed (code INB) in its 2012 handbook for the Bachelor of Information Technology (QUT 2012).

- Teamwork, communication skills, integrity, reliability, and self-motivation were reported by Wong, von Hellens, and Orr (2006) to be more important to employers than purely technical skills.
- A survey on Employer Satisfaction with Graduate Skills (DETYA, 2000) for all discipline areas rates enthusiasm, motivation, independence and critical thinking abilities as of paramount importance in graduates. Academic achievement was used as an indicator of such things as motivation, problem-solving ability, and learning capacity.
- Hagan (2004) found that ICT employers were most often dissatisfied with the graduates' project management abilities, lack of understanding of business processes, poor written communication skills, and the standard to which they were able to interact with clients.
- Koppi, Sheard, Naghdy, Chicharo, Edwards, Brookes, and Wilson (2009) examined the graduate perspective and found that graduates felt technically competent but lacked interpersonal and business skills.

The main stakeholders in tertiary education are: the students, the employers, and the educators themselves. Noting that students must believe in the relevance of their courses to their future employment (Nagarajan & Edwards, 2008) an approach to curriculum review is needed which yields the best outcomes for all.

The questions of what is required and how best to provide it remain. Gruba, Moffat, Sondergaard, and Zobel (2004) tentatively conclude that curriculum change in universities is rarely best influenced by the educators. They present a picture of curriculum review being driven by dominant individuals with change motivated by financial concerns, academic fashion, and student interest. Pedagogical concerns only influence change at the micro (unit) level.

An objective framework is thus needed to guide the curriculum review. The Australian Computer Society (ACS) in its accreditation manuals (ACS, 2011) provides the following:

1. Identify potential ICT roles that could be undertaken by graduates of a given program of study.
2. Identify the skills required by professionals in a given ICT career role.
3. Identify the level of autonomy and responsibility developed.
4. Identify the ICT Role-Specific Knowledge required to practise the skills.
5. Identify Complementary Knowledge that supports the skill set or that broadens student employability.
6. Design a course structure that incorporates ICT Role Specific Knowledge with the Core Body of Knowledge and other Complementary Knowledge as part of a holistic program of study.
7. Collect artefacts to demonstrate that skills have been developed by students to an appropriate level.

The new degree at UTAS is essentially a completely original degree and its design is being undertaken with a 'green-fields' approach. It is UTAS' intention that the degree receive ACS accreditation (as previous offerings have) and hence adopting the ACS framework seems a reasonable and beneficial approach. More information on the adoption of the framework is available in (Herbert, de Salas, Lewis, Cameron-Jones, Chinthammit, Dermoudy, Ellis, and Springer 2013).

3 Gaining feedback

3.1 Approach

The approach has been decomposed to well-defined stages. The first stage has been to gather information so that tentative degree design down to the unit level can occur. This degree design is the second phase; at the time of writing it is nearing completion. The third phase is to re-engage the industry members to gain feedback and endorsement of the tentative design, before finally presenting the degree design to UTAS' senior committees and Academic Senate for approval prior to delivery.

In order to obtain as much information as possible and to allow particular avenues of enquiry to be pursued, face-to-face communication was undertaken through forums and interviews. At each forum the attendees were divided into small groups of approximately three or four attendees. Where there were multiple such groups, each group moved around a set of interviewers, so each interviewer was asking the same set of questions to each group; each group spent approximately twenty minutes with each interviewer.

3.2 Participants

The forums and interviews comprising the first stage of curriculum design were promoted as an opportunity to influence the future ICT direction of the State. Stakeholders included ICT industry representatives together with pre-tertiary and TAFE educators from institutions across Tasmania.

3.2.1 Industry participants

The number of people seeking to participate was overwhelming; for logistical purposes the number of participants was limited to twenty-odd industry representatives. Approximately ten members of Tasmania's ICT industry could not be included in the first round of information gathering due to their work commitments at the time; these people will be included in the next stage — stakeholder feedback on the new degree proposal.

Representatives of local and national ICT industry and government participated in the interviews. More specifically, the types of organisations represented included those involved in:

- IT Recruitment
- IT Service/Consulting
- IT Information Management
- IT Security
- Research and Development
- Software development
- Hospitality/Tourism/Gaming/Transport/Retail
- Fishing Aquaculture/Food Processing

- Engineering
- Education
- Government (Federal and Local)

ICT businesses of all sizes were represented. There were a number of medium sized software development companies where the number of ICT employees ranged from 20–30, there were some businesses (including government departments) that had a number of ICT employees ranging from 1–40, and there were four large national organisations with in excess of 150 ICT employees, some with thousands.

All the interviewees (except one who was a recent graduate) had employed graduates into various positions throughout their career; most had employed fewer than ten, but some had employed as many as fifty or more.

3.2.2 Education participants

Most UTAS students are matriculants who have completed the Tasmanian Certificate of Education (TCE). A significant number of students also enter UTAS from the Polytechnic² and Skills Institute with completed cognate Advanced Diploma, Diploma, Certificate IV and Certificate III awards. In an attempt to ensure articulation with pre-University educational awards and institutions, and to avoid duplication of content delivery and skill development a number of stakeholder educational institution representatives were also involved.

Representatives from the government and non-government college sector (Academies and Polytechnics) participated together with representatives from the Skills Institute.

3.3 Question themes

For each industry interview or forum the same set of questions was used. These questions sought information about:

- The desired skill set — including both the ‘soft-skills’ and domain-specific technical skills — a graduate should have.
- The participant’s perception of the “added value” of a university graduate compared to a TAFE/Polytechnic/Skills Institute graduate.
- Desired career outcomes for graduates of bachelors degrees and coursework postgraduate degrees.
- Topics which should be included in the degree.
- The nature of likely graduate-entry positions in the next five years.

Similar questions were asked of the educators. In particular:

- The participant’s perception of the distinction between a university graduate and a TAFE/Polytechnic/Skills Institute graduate.

² In Tasmania, high schools teach years 7–10 and colleges teach years 11–12. Government colleges comprise schools with a focus on academic achievement (“Academies”), academic and vocational achievement (“Polytechnics”), and vocational achievement (“Skills Institutes”). TAFE colleges were recently re-integrated with colleges to yield the Polytechnic and the Skills Institute.

- Topics which should be included in the degree.
- Information on their course offerings and for pathways and collaborations.

4 Messages

The information collected from the interviews and forums was analysed at a thematic level in order to identify key messages. The broad themes are presented below along with specific advice from both industry and educators.

The analysis of the interviews and the forums from industry resulted in information relating to technical skills, soft skills, core knowledge, work integrated learning, and degree /diploma perspectives. The analysis of the interviews and forums from the educators resulted in information relating to offerings based on interest, TCE, vocational versus university perspectives, entry and articulation, gender appeal and retention, and working together. A summary of each will now be presented.

4.1 Advice from industry

4.1.1 Technical skills

Interviewees indicated that the technical skills of graduates should include both the ability to develop (‘build’) and the ability to operate (‘use’/‘do’). Graduates should have basic programming skills but this competence wasn’t requested in any specific programming language. The general opinion was that as long as the chosen programming language was taught in depth, the ICT graduates would be able to adapt quickly and effectively to another language. The interviewees expressed a desire that an ICT graduate should have at least one in-depth technical competency area, such as (but not limited to) software development, which was supported by a broad range of context and application knowledge of ICT. The identification of the specific technical skill was unimportant, but the capacity to develop one was said to be essential.

Other skills/knowledge mentioned by industry participants as being necessary in graduates included:

- Those from fundamental computing and traditional computer science such as formal methods, basic mathematics, logic, and the history of ICT.
- An understanding of data structures and databases.
- Greater capability in the use of Microsoft Office applications. Excel skills in particular were mentioned and an expectation was voiced that ICT graduates should have such skills possibly to an advanced level.
- Those related to application development for mobile devices which was considered to be increasingly main-stream.
- Ensuring graduates could function in varying operating system environments. This may be in response to the fact that UTAS is a strong partner in the Apple University Consortium and many industry members expressed their feelings that the School is too Apple focused.

In general, employers of past graduates who were interviewed were happy with the ICT technical skills currently being demonstrated by graduates. They were

content to teach specific ICT skills on the job. This is a change from a few years ago when we last consulted with industry at which time they were insisting on specific tools and technology to be taught at the University.

4.1.2 Soft skills

Generic abilities such as communication skills were acknowledged as being of equal importance for UTAS to teach as technical skills. The interviewees were insistent that the University should produce professionals with the ability to communicate; it was suggested that this is the current added value that a university graduate has over non-university graduates. The interviewees recommended taking the students out of their comfort zones by making them do presentations and debates to develop improved communication skills.

Additionally, interviewees expressed the need for graduates to appreciate the drivers of business, and to be able to undertake the related activities of analysis, modelling, business process management, and project and change management.

Interviewees identified that there was no longer room in the industry for graduates who could not relate well to business and clients. Employers want graduates to have a broad ICT knowledge so they have the ability to understand the needs of clients or users. It was felt that graduates can mature into a particular area later. Those who are too specialised are unlikely to be chosen over a graduate with a broad range of ICT skills for the Tasmanian industry.

4.1.3 A core body of knowledge

Interviewees preferred that students were taught principles and context, rather than specific development languages and current tools (although Microsoft Office products were an exception). Employers believed that they could teach new tools and languages to those who understood what features tools and languages provided, but graduates needed to understand principles to allow them to continue to learn and adapt as technology emerged and evolved.

Industry indicated for graduates to be useful, they need to understand how all the ICT content links together. This sort of understanding helps with analysis and understanding the needs of clients.

4.1.4 Real work experience

Most of the employers interviewed only employed university graduates and almost exclusively ICT graduates. Local businesses predominantly hire UTAS graduates. The ICT subjects completed as part of the degree were not of major interest to employers, however, employers wanted evidence that graduates had completed enough ICT technical subjects in their degree. They compared the overall results of graduate candidates for a position and then decided who to employ at the interview based on communication skills, personality and ability to fit in with existing employees.

Industry desired graduates that have had real job experience, "even if it is just at a fast-food restaurant" was a comment often made. Many interviewees had participated as clients in the existing capstone project units, and they thought that this was good for developing

team work and client interaction skills but work-integrated learning through workplace placements would provide a better understanding of what to expect on entry into the workforce and would allow the development of additional skills, such as customer awareness. They also recommended that industry participate more in the teaching program to bring in real world examples and industry perspectives to the material being taught.

4.1.5 Vocational versus university qualification

Interviewees see a distinct difference between TAFE (certificate IV, V) and University graduates. TAFE graduates are seen as more practical and employed for specific limited tasks. University graduates are considered to have a broader spectrum of skills and knowledge. University graduates have the ability to think critically and approach tasks with a broader view, they are believed to be generally more mature and by completing a degree they have demonstrated the ability to "stick at something". Although most interviewees initially employed graduates as software developers or in help desk type support roles, the university graduates are then promoted to software analysts, designers, or system administrators.

It is clear from these views that a mixture of theory and practical capability needs to be retained in the new degree and that students need to continue to be taught the generic graduate attributes of, in particular, problem solving, critical thinking, and life-long learning.

4.1.6 Other useful insights

The issue of an increased use of outsourcing and off-shoring was identified as a possible impact on graduate software developer positions. "*No matter how good our programmers are they are too expensive to compete with India's outsourcing*" was the advice of one interviewee. Graduates need business analysis, sourcing and integration, and project management skills as this is their future.

Many interviewees said that they did not employ, or even interview, Bachelor of Information Systems graduates as it was perceived that these graduates did not have enough technical skills necessary for the types of initial roles that ICT graduates undertake. Interestingly, many interviewees said business analysis skills combined with technical skills were important and in general they were not happy with the business acumen of Bachelor of Computing graduates. Graduates need an understanding of business structure and practices.

Many interviewees were concerned about the standard of the weaker graduates, with a suggestion to raise the entry bar to the degree. Industry representatives were equally concerned by the decreasing number of students graduating from the ICT degrees and the fact that demand for graduates is exceeding supply. Discussion also took place on the quality of the international graduates and in particular how to improve their communication skills and hence make them more employable in the local industry.

When asked, the interviewees liked the idea of having a single degree with reduced options and a few clear majors as this would remove confusion and ensure all ICT graduates had a balance of technical and non-technical skills.

During the interviews there was discussion about the future of ICT both in the State and nationally. Industry identified that the future direction of ICT is not easy to predict but future ICT jobs will still require fundamental ICT skills.

In summary, there was a very strong emphasis on producing professionals with generic skills such as verbal and written communication, team work, ability and desire to learn, and problem solving along with non-technical skills in areas such as requirements analysis, business analysis, project management, and sourcing and integration. In addition a broad base of technical ICT fundamental skills are required including programming, data structures and databases, mobile and web programming, and low level tool skills — in particular Microsoft Office products. The interviewees from industry were in favour of an “all-rounder” rather than aiming for a specific targeted career outcome.

4.2 Advice from educators

The purpose of the educator forums was to identify what ICT background students have before coming to UTAS, what UTAS should value-add to graduates, and what pathways/incentives/content could be established to encourage more students to enrol.

4.2.1 Offerings based on interest

Non-government college-level ICT training is now focusing on developing the specific interests of students, rather than preparing them for job-readiness in ICT. As a result, traditional ICT training in Computer Science and Information Systems was less likely to be taught in the future; newer attractive topics like Computer Graphics, Games, AI, and Robotics are being introduced. Similar changes are occurring in Government schools with Computer Graphics enrolments far exceeding those in the pre-tertiary subjects of Computer Science or Information Systems.

Interviewees suggested that the students tend to want to learn more in the areas they are already aware of. There was a general feeling that University should offer a broad range of topics to meet student interest and to give them a rounded introduction to many topics.

4.2.2 Tasmanian Certificate of Education

The college curricula for Computer Science and Information Systems in Tasmania are being redeveloped for 2013 and the introduction of the national curriculum in high school and college makes it harder to prepare a university curriculum that builds on existing lower levels of ICT education.

At college, a technology subject is mandatory but this does not necessarily need to be a university pathway (TCE) subject. College interviewees recommended that entrants receive reward or credit if they have completed Computer Science or Information Systems TCE subjects, as an incentive to enrol at university. This was also seen as a possible way to increase enrolments at college in these subjects that currently have low student interest and enrolments compared to the more popular, but less rigorous non-TCE topics like Computer Graphics.

To increase enrolments at university it was identified that there was a need to tap into the growing Computer

Graphics numbers at college. Many students choose this as their college elective to complement their main interest area. It is seen as something interesting and relevant to many fields.

4.2.3 TAFE versus university

Polytechnic ICT teaching is focused on developing graduates to specific operational ICT jobs. Non-TCE offerings that support programming, web technologies, system administration, databases, and networking, have been recently revised in the light of industry demands. It is therefore important that the new degree retains flexible entry and not rely on TCE subjects.

When asked “why do students want to do ICT at university?” the interviewees indicated students come to university to do more ICT (not maths or business). They urged us to ensure that they get enough of what they want. They come expecting it to be more interesting material than what they get at college or Polytechnic. They come because they think on graduation they will get better jobs than they would with just a college or Polytechnic qualification. After a tour of our facilities it was mentioned repeatedly that the “hack space” (a room where students can do their own thing and explore ideas with software and hardware) was something that students would enjoy.

Polytechnic representatives saw the difference to University being that University should teach similar skill sets, but in much greater depth, surrounded by much broader context.

4.2.4 Entry and articulation

Educators feel the degree entry requirement should be sufficiently low that it attracts a broad range of students coming from the Polytechnic and college. However, this view is contradicted by industry members who want a higher entry requirement in order to get a higher quality graduate.

Clear articulation with the Polytechnic was recommended so these students can build on their prior learning rather than just repeat their college experience with more theoretical material added — which is not of great interest to them.

Interviewees indicated they had received feedback from their students who had gone onto university that some of the first year content was identified as being too low-level for some of the top students coming from the colleges and Polytechnic. The perception is that they have done it before. Interviewees recommended investigating offering credit (or different options) for top students to encourage transition. Catering for the top students was identified as something that should be a priority because at the moment they are leaving university as they are bored and feel they are simply doing the same material again.

4.2.5 Girls versus boys

Interviewees indicated their male students wanted hands-on activities in their learning. More practical components, especially early on will attract and retain new entrants much better. Advice from the Polytechnic was to keep the students interested and engaged, so that they did not focus upon the two or three years that they

had ahead of them. The view was that they do Polytechnic courses as they are practical and shorter.

The lack of interest from female students, who anecdotally tend to be less interested in the hands-on approach, is a concern for all levels of ICT education.

4.2.6 Working together

There was recognition from all interview participants that industry wants all levels of the education system working together to produce the best ICT personnel possible to meet the ICT needs of Tasmania. It was clear that the college and Polytechnic educators needed the university degree to be attractive as it would attract students to ICT at their levels. They strongly encouraged more collaboration to ensure understanding of each other's offerings and directions and to create attractive pathways. ICT college teachers would like the University to host professional development days as there are no longer moderation days, so teachers do not have the opportunity to get together any more.

5 Outcomes for curriculum design

The initial degree framework has been completed and has been guided by the input received from industry and educators.

The result is a three-year (24 unit) degree with three different majors comprising much of the ACM curricula: "ICT Professional" (which will be compulsory), "Software Development", and "Games and Creative Technology" (which continues the teaching in UTAS' successful Games Technology and Human Interface Technology sub-disciplines). Students will complete eleven technical units with four units at different levels focused on professional and non-technical skill development. The structure ensures that all students have the technical, non-technical, and generic professional skills needed, while — with the inclusion of at least four units that are electives from any discipline — still affording enough choice to further their individual interests.

The following points outline the key components for the proposed degree:

- Professional skills units are to be introduced from first year to develop communication and teamwork skills early. Further development of teamwork and communication skills have been included at all levels to provide depth in professional skills for the students.
- Breadth in ICT topics will be introduced through the range of units provided. Depth has been created with units to be offered at all year-levels in a hierarchy requiring pre-requisites and with integrated content. In previous offerings, units were perceived as being disjointed (silos) and consequently there was little opportunity to develop a depth of knowledge. Broad ICT topics will now be more practically oriented (while maintaining the theoretical content and professionalism necessary to differentiate this topic from what is offered at the Polytechnic).
- There is depth in software development with compulsory units at all three levels. There is one compulsory programming unit at first-year and

three programming units at second year when the bulk of the students are more developed and able to cope with the material. The compulsory first-year programming unit will have a prerequisite, ensuring all students have some programming experience on entry. The requirement for pre-requisite knowledge will facilitate a more interesting and challenging unit — which should have a positive impact on retention. Flexible options at degree and pre-degree level will be available to qualify students that do not have prior programming experience.

- In response to the demands for business acumen in our graduates all students will be required to complete units in entrepreneurship, project management, requirements and business modelling, business analysis; and system sourcing and integration to develop non-technical ICT skills.
- ICT career outcomes and the relevance of the content of the degree to those outcomes will be mapped out in a first year unit to give students a context for what they can achieve as an ICT professional. Consideration is being given to introducing elective industry-based experience units at all levels to enable students to gain genuine work experience throughout their degree in addition to the capstone experience they receive in the final year project units. There is an intention to embed talks by industry speakers throughout all units to relate the content of each unit to what the students will experience in employment.
- Artificial Intelligence, a key research direction of the School, has been proposed as a first year unit in an attempt to inspire students to consider a research career. This unit will be accessible to a broad range of university students and will be developed to appeal to non-ICT students who are just as likely to be intrigued by the topic matter.
- Units that relate to research focuses in the School will also be offered but will have pre-requisites that restrict enrolment to the top students. Small classes and special experiences for these elite students will hopefully inspire in them a desire to stay to complete higher degrees and pursue a career in research or innovation in the field of ICT.
- There is an intention to embed 'research hotspots' throughout all units and relate the content of each unit to research that is happening in the field. Consideration is being given to introducing specialised R&D units at all levels for the elite students, so that research experience can be obtained from an early stage.
- A compulsory bridging unit will be introduced for all international students given a year or more credit. Many international students are given advanced standing on the basis of prior learning in their own country. This results in them being 'slotted' into second or third year at UTAS and by-passing units that have provided

incidental induction to UTAS and which have developed communication and team work skills. As a result many are technically competent but are not best able to participate in group work and compete for employment on graduation. The bridging unit will redress these deficiencies and ensure all students are at a high standard on graduation with well developed communication and teamwork skills.

- Two units with a focus on visualisation are being considered for first year to attract the students who were interested in Computer Graphics. One will focus on the visualisation of information and the other on visualisation as it relates to computation and simulation.
- Two pervasive themes are to be developed throughout the degree: security and user-centeredness including HCI. The pervasive themes alongside the three depth areas (software development, professionalism, broad ICT knowledge) and the need to present the content from both a technical and non-technical perspective will discourage teaching in silos. The approach will focus on relating the content of each unit to other units in the degree. This will generate graduates who have a much better understanding of the relationship between the ICT content.

6 Conclusion

A role of universities is to educate and produce 'well rounded' graduates who meet industry demands for job readiness. Having been provided with the rare opportunity to design a 'green-fields' degree, the School looked to engage with key stakeholders such as industry and other educators for advice. Curriculum review guidance was sought from the ACS and this paper has explored the second stage of the review process.

The key messages that have been incorporated into the design and structure of the new degree are:

- ICT graduates need generic professional skills and non-technical skills in addition to technical ICT skills and tool skills.
- An ICT graduate should have at least one in-depth technical competency area.
- An ICT graduate needs a broad range of ICT knowledge. Employers believe it will give graduates the ability to understand the needs of clients/users. Educators believe a broad range will attract more students into the degree.
- There is no longer room in the industry for ICT graduates who are unable to relate well to business and clients. Graduates need business analysis, sourcing and integration, and project management skills as this is their future as a result of the growth in outsourcing and off-shoring.
- UTAS should produce the best ICT personnel possible to meet the ICT needs of Tasmania.

The School is currently refining and documenting information relating to the structure and design of the new degree. The documentation will provide information on the proposed content of the units in the degree. Once

completed, the School can move to stage three of the process — to re-engage with key stakeholders for feedback and hopefully endorsement.

7 References

- ACM, Association for Computing Machinery (2008), Association for Computing Machinery (2008): *Computer Science Curriculum 2008*, <http://www.acm.org/education/curricula/ComputerScience2008.pdf>. Accessed 17 August 2012.
- ACS, Australian Computer Society (2011), Australian Computer Society (2011): *Accreditation Manual*, ACS.
- DETYA, Department of Education, Training and Youth Affairs (2000), *Employer Satisfaction with Graduate Skills Research Report*, Commonwealth Government of Australia.
- Gruba, P., Moffat, A., Søndergaard, H., & Zobel, J. (2004), "What Drives Curriculum Change?" in *Proceedings of the Sixth Australasian Computing Education Conference*, pp 109–117, ACS.
- Hagan, D. (2004), "Employer Satisfaction with ICT Graduates" in *Proceedings of the Sixth Australasian Computing Education Conference*, pp 119–123, ACS.
- Herbert, N., de Salas, K., Lewis, I., Cameron-Jones, M., Chinthammit, W., Dermoudy, J., Ellis, L., and Springer M. (2013), "Identifying career outcomes as the first step in ICT curricula development". Accepted in the *Fifteenth Australasian Computing Education Conference (ACE2013)*, Adelaide, Australia, January 2013.
- Ignite, (1999), *Skills in Demand*, <http://www.ignite.net.au/skills/> Cited in Orr & von Hellens (2000).
- Koppi, T., Sheard, J., Naghdy, F., Chicharo, J., Edwards, S., Brookes, W., & Wilson D. (2009), "What Our ICT Graduates Really Need from Us: A Perspective from the Workplace" in *Proceedings of the Eleventh Australasian Computing Education Conference (ACE2009)*, Wellington, New Zealand, January 2009.
- Monash University (2012), *2012 Handbook*, <http://monash.edu/pubs/2012handbooks/units/index-byfaculty-it.html>, Accessed 8 Aug 2012
- Nagarajan, S. & Edwards, J. (2008), "Towards Understanding the Non-technical Work Experiences of Recent Australian Information Technology Graduates" in *Proceedings of the Tenth Australasian Computing Education Conference*, pp 103–112, ACS.
- Orr, J. & von Hellens, L. (2000), "Skill Requirements of IT&T Professionals and Graduates: An Australian Study — Research in Progress" in *Proceedings of the 2000 ACM SIGCPR Conference on Computer Personnel Research*, pp 167–170, ACM.
- Queensland University of Technology (2012), *2012 Study Handbook*, <http://www.qut.edu.au/study/courses/bachelor-of-information-technology>, Accessed 8 August 2012
- Swinburne University of Technology (2012), *2012 Courses Handbook*, <http://courses.swinburne.edu.au/courses/Bachelor-of-Information-Technology-I050/local>, Accessed 8 August 2012.
- The Australian Newspaper (2006), *Articulate Workers Wanted*, 01 November 2006, Viewed online at

www.highbeam.com/doc/1G1-153601934.html
8/8/2012.

UTAS (2012), *Courses and Units 2012 Handbook*,
[http://courses.utas.edu.au/portal/page/portal/COURSE
UNIT/UTAS_CU_ENTRY?P_YEAR=2012&P_CONT
EXT=NEW](http://courses.utas.edu.au/portal/page/portal/COURSE_UNIT/UTAS_CU_ENTRY?P_YEAR=2012&P_CONT_EXT=NEW), Accessed 17 August 2012.

Wong, S., von Hellens, L., and Orr, J. (2006), "Non-technical skills and personal attributes: the Soft Skills Matter Most" in *Proceedings of the Sixth Australasian Women in Computing Workshop*. Cited in Nagarajan & Edwards (2008).

Measuring the difficulty of code comprehension tasks using software metrics

Nadia Kasto and Jacqueline Whalley

Software Engineering Research Laboratory
 School of Computing and Mathematical Sciences
 AUT University
 PO Box 92006, Auckland 1142, New Zealand
 {nkasto, jwhalley}@aut.ac.nz

Abstract

In this paper we report on an empirical study into the use of software metrics as a way of estimating the difficulty of code comprehension tasks. Our results indicate that software metrics can provide useful information about the difficulties inherent in code tracing in first year programming assessment. We conclude that software metrics may be a useful tool to assist in the design and selection of questions when setting an examination.

Keywords: software metrics, code comprehension, novice programmers, assessment.

1 Introduction

It is common knowledge that novice programmers find programming particularly difficult and that assessing the knowledge and skills the students have gained is problematic. Historically the pass rates for students undertaking first year courses have been relatively low. This in part might be due to some difficulties related to the assessment of these courses. Whalley et al. (2006) noted that “assessing programming fairly and consistently is a complex and challenging task, for which programming educators lack clear frameworks and tools” (p. 251). More recently, Elliott Tew (2010) suggested that “the field of computing lacks valid and reliable assessment instruments for pedagogical or research purposes” (p.xiii).

In order to write better questions and assessments computer science educators have attempted to apply various educational taxonomies to guide the design of assessments. In 2006 an analysis of a program comprehension question set within two key pedagogical frameworks: the Bloom (Anderson et al. 2001) and SOLO (Biggs and Collis 1982) taxonomies was reported (Whalley et al. 2006). It was found that student performance was consistent with the cognitive difficulty levels, indicated by the assigned Bloom category of the questions. Additionally a degree of consistency was found between student performance and the SOLO taxonomy level of their responses to an ‘Explain in Plain

English’ (EiPE) question. While these results and results of subsequent studies by the Bracelet project team were encouraging (e.g.: Lister et al. 2006, Thompson et al. 2008, Clear et al. 2008, Sheard et al. 2008, Whalley et al. 2011) many educators have reported difficulties in reliably using these and other taxonomies in the context of novice computer programming assessment design, evaluation and research (e.g.: Fuller et al. 2007, Thompson et al. 2008, Shuhidan, Hamilton and D’Souza 2009, Meerbaum-Salant, Armoni and Ben-Ari 2010)

An alternative or supplementary approach to informing the assessment instrument design process might be to use software metrics in order to determine the difficulty of examination questions designed to assess novice programmers.

2 Background

Typically research into software metrics is conducted in the context of relatively large scale commercial software development projects. However some work using software metrics to support research related to the improvement of teaching and learning of computer programming has been undertaken.

One study applied software metrics to previously reported code used in empirical studies of novice and expert program comprehension (Mathias et al. 1999). The metrics were used in order to examine the underlying nature of code designed to study the process of program comprehension. The software metrics used in this study were *lines of code* and *cyclomatic complexity* (McCabe 1976). A correlation was found between the complexity of the code and the comprehension strategies observed by the original researchers suggesting that *lines of code* and *cyclomatic complexity* might correlate to the difficulty of small program comprehension tasks.

Parker and Becker (2003) employed Halstead’s metrics (Halstead 1977) to measure and compare the effectiveness of students solutions of two different code writing assessments based on the premise that the metrics can be seen as a measure of work done. An earlier empirical study measuring student solutions to code writing questions using software metrics and comparing those measures with student performance found that neither *lines of code* nor Halstead’s metrics were able to predict the error rate in the student’s solutions (Klemola 1998). Subsequently, Klemola and Rilling (2003) developed a software metric called the *Kind of Line of Code Identifier Density* (KLCID) metric for analysing the cognitive complexity of program comprehension tasks. KLCID was designed to capture the effect of the number

of unique kinds of code lines in a program segment. For KLCID only conceptually unique lines of code are counted and within these unique lines the identifier density is calculated (Klemola and Rilling 2003). The effectiveness of the KLCID metric was evaluated in a study of code comprehension tasks from a final examination of an introductory C++ course. The complexity of each task as measured by KLCID was compared with the average student performance on the task. A correlation was found between increasing KLCID and decreasing student performance. This finding is not surprising as in text comprehension it has been found that a higher density of concepts decreases the rate of comprehension (Kintsch and van Dijk 1978). The authors concluded that KLCID was “a good candidate to measure the complexity of code comprehension assessment tasks within the same course” (Klemola and Riling 2003). However the code comprehension examination questions themselves are not reported so it is difficult to determine

the general applicability of the KLCID metric to novice programmer code comprehension tasks.

3 Software Metrics

In order to attempt to measure the difficulty of typical code comprehension and code tracing examination questions we first selected an appropriate set of software metrics. Software metrics focus on a particular feature of a program and are often devised with a single programming paradigm in mind. Table 1 shows a set of commonly employed software metrics classified by metric type and their applicability to three programming paradigms.

The examination questions that we have analysed are from a CS1 (first semester) Java programming course. The questions are typical code tracing and EiPE questions that have been reported extensively in the recent literature (e.g.: Venables, Tan and Lister 2009, Murphy, McCauley and Fitzgerald 2012).

Metric Type	Metric	Programming Paradigm		
		imperative	structural	object oriented
Basic	Number of lines of code	✓	✓	✓
	Number of <i>blank</i> lines of code	✓	✓	✓
	Number of comment lines of code.	✓	✓	✓
	Number of comment words.	✓	✓	✓
	Number of statements	✓	✓	✓
	Number of methods.		✓	✓
	Average line of code per method.		✓	✓
	Number of parameters.	✓	✓	✓
	Number of import statements.		✓	✓
	Number of arguments.		✓	✓
	Number of methods per class.			✓
	Number of classes referenced.			✓
	Average number of attributes per class			✓
	Number of constructors.			✓
	Average number of constructors per class.			✓
KLCID	✓	✓	✓	
Complexity metrics	Cyclomatic complexity	✓	✓	✓
	Nested block depth.	✓	✓	✓
Halstead metrics	Number of operands.	✓	✓	✓
	Number of operators.	✓	✓	✓
	Number of unique operands.	✓	✓	✓
	Number of unique operators.	✓	✓	✓
	Effort to implement.		✓	✓
	Time to implement.		✓	✓
	Program length.		✓	✓
	Program level.		✓	✓
	Program volume.		✓	✓
	Maintainability index.		✓	✓
Object oriented	Weight method per class.			✓
	Response for class.			✓
	Lack of cohesion of methods.			✓
	Coupling between object classes.			✓
	Depth of inheritance tree.			✓
	Number of children.			✓

Table 1: Static metrics and their applicability across programming paradigms

Although the course is taught with an objects first approach most of the comprehension questions are small bite size pieces of code and are largely procedural. Therefore, even if the code is encapsulated in a method, many of the questions are essentially procedural in nature.

Of the metrics in Table 1 we selected the subset which we deemed to be most applicable to measuring the difficulty of novice code tracing and EiPE tasks:

- Number of statements
- Number of operands (including all identifiers that are not key words)
- Cyclomatic complexity
- Average nested block depth
- Average number of parameters

One EiPE question involved code that contained two methods and internal method calls. The object oriented metric, the *number of methods*, that had a variation in value was therefore included as part of our metric set for EiPE questions.

We did not use KCLID because most of our code comprehension questions did not contain lines of code which were not conceptually unique lines of code. Additionally, we elected not to use the *number of operators* metric as the *number of operators* is proportional to the *number of operands* and would therefore not contribute anything new to the evaluation.

We also supplemented this set of metrics with two simplified versions of dynamic metrics for the measurement of the difficulty of the code tracing questions that we have called the *sum of all operands in the executed statements* and the *number of executed program statements*. The *sum of all operands in the executed statements* was calculated as the sum of all operands (O) in the executed statements ES where the total number of executed statements is V .

$$\text{Sum of all operands in the executed statements} = \sum_{i=1}^V ES_i(O)$$

The *number of executed program statements* was counted as the total number of statements executed for the complete tracing task. This count, if a selection or iterative statement is included in the code, is dependent

on the data provided as the input for the specific tracing task.

These dynamic metrics provide a measurement of the execution complexity of the code. It seems reasonable to include such metrics because when students are tracing code they are hand executing the code and, from an initial input, processing data through the code line by line via the relevant paths of the code in order to determine the output. We postulate that these metrics will correlate well with the difficulty of the tracing task.

4 Data Sets

The questions analysed in this study were selected from several occurrences of a final examination for a first year Java programming course. The teaching team and pedagogy was the same for all instances of the course and the results were taken from exam scripts for which the students had given ethical consent for their data to be used. These students were representative of the entire cohort.

For the code tracing questions two examinations were analysed. One examination contained the questions 1A-D and resulted in 93 student responses for analysis and the other contained questions 2A-2E for which 79 student responses were analysed (Table 2). The EiPE questions were selected from three examinations. For 3A-D, 4A-C and 5A-E there were respectively 93, 79, and 92 student responses analysed. The percentage of fully correct answers is used as the measure of question difficulty.

The distribution of the percentage of fully correct answers was irregular and clustered. We therefore used natural, data driven, clustering to place the data into a five point scale from very easy (a relatively high percentage of students got the question correct) to very hard. Questions of similar difficulty, as determined by student achievement, for example 1D (26%), 2D (21%) and 2E (27%) were therefore ranked at the same difficulty level. These ranks were then used to determine whether or not there was a correlation between difficulty and the relevant metrics. It seemed unlikely that one common set of software metrics would provide useful information about different types of questions or about questions designed to measure significantly different types of knowledge. For this reason the data from the code tracing and EiPE questions were placed in separate data sets.

	questions								
	1A	1B	1C	1D	2A	2B	2C	2D	2E
Difficulty ranked	1	4	4	8	4	6	2	8	8
Cyclomatic complexity	1	2	2	3	1	1	3	2	3
Average nested block depth	1	2	2	3	1	1	2	2	3
Number of operands	14	10	12	29	13	5	13	12	17
Number of parameters	0	2	1	2	0	2	3	1	1
Number of statements	7	5	5	8	3	1	2	4	3
Sum of all operands in the executed statements	14	18	33	48	13	10	35	42	138
Number of commands in the executed statements	7	9	13	52	6	4	13	20	34

Table 2: Metrics for code tracing questions

An item discrimination analysis was undertaken to examine the relationship between student scores for each question and the total score for the related set of questions to identify any outlier questions that did not therefore belong in the data set. A point bi-serial correlation was calculated between each question and the total score, excluding the score for the question itself, for all questions in that set (tracing questions or EiPE questions). This provides an estimate of the extent to which an individual question is measuring the same competencies as the rest of the questions in that question set. Each question is expected to contribute to the total score for that question set. Any question that does not correlate positively with the total score is probably measuring something other than what the examiners intended and does not belong in that set.

For all questions except 2C and 2B a significant positive correlation was found between students' scores on the question and their overall scores on the related set of questions. Therefore, except for 2C ($r_{pb} = 0.165$, $p = 0.15$) and 2B ($r_{pb} = 0.217$, $p = 0.06$) the questions in each set are contributing towards the respective total scores and can be considered to belong within the sets. However, the discrimination analysis also provides evidence that for some reason 2C and 2B are not measuring the same thing as the other code tracing questions. Therefore, for the purpose of further analysis, we removed both of these outliers from the data set.

The students found 2C relatively easy while we would have expected that this would be one of the more difficult code tracing questions. The students had been introduced to this code in lectures and had been guided through a similar tracing exercise with slightly different input data. Perhaps this is encouraging; clearly teaching has had some impact on student learning. Nevertheless, if test questions are set that are too close to specific examples taught in lectures they may be measuring the students' abilities to remember specific examples rather than measuring their code tracing abilities. That is, they may well be measuring something other than what the examiner intended.

On the other hand, we would have expected question 2B to be an easy question but student performance showed that they found it to be relatively difficult. Question 2B is a simple method that calculates the remainder. We believe that the issue in this question may

lie with a lack of mathematical knowledge rather than a lack of programming comprehension. This conjecture is supported by the fact that many of the same students were able to answer code tracing questions that consisted of more complex code successfully. Once again it seems that the question is not measuring what the developers intended and does not belong in the data set.

5 Results

The code provided in the examination was analysed using our set of software metrics. In the case of the dynamic metrics for the code tracing questions the metrics are calculated from those parts of the code that are executed in order to arrive at the correct answer. We then compare the metrics with the student performance on the questions. The following metrics were calculated using the Rationale® Software Analyzer 7.1 (RSA 2012) tool: *number of operands*, *cyclomatic complexity*, *average nested block depth*, *average number of parameters*, and *number of methods*. Initially we calculated *lines of code*, using Rationale® Software Analyzer, as the total number of executable lines of code. In the programming examination questions the code is formatted so that the opening and closing braces are placed on their own line. Given the small size of the code for each question, lines containing only braces contribute significantly to the *lines of code* metric when calculated this way. We believe that these lines do not contribute to the complexity or difficulty of the code comprehension tasks. Consequently, we calculated the *number of statements* rather than the total lines of code.

The significance of the correlation of each metric to the categorised difficulty (encoded numerically where the easiest is ranked as 1) of each question was then tested using Kendall's τ -b. Kendall's τ -b was chosen because the datasets contained tied ranks. Table 4 gives the Kendall's τ -b for all the, tracing and EiPE, questions analysed.

It is worth noting that the tracing and EiPE exam questions used in this study are characterised by a low number of number of program commands and are generally confined to one or two methods. As a consequence the *cyclomatic complexity* for the exam questions does not exceed 5 and the *nested block depth* does not exceed 3.

	questions											
	3A	3B	3C	3D	4A	4B	4C	5A	5B	5C	5D	5E
Difficulty ranked	8	10	8	3	5.5	11.5	11.5	1	5.5	3	8	3
cyclomatic complexity	1	2	3	4.5	2	3	5	2	3	2	3	1
Average nested block depth	1	2	3	2.5	2	3	3	2	3	2	3	1
Number of operands	11	11	18	37	14	21	36	11	21	18	39	6
Number of parameters	0	2	2	2	1	1	2	2	1	2	1	0
Number of statements	5	5	6	11	5	5	16	5	5	5	9	3
Number of methods	1	1	1	2	1	1	1	1	1	1	1	1

Table 3: Metrics for 'Explain in plain English' (EiPE) questions

For code tracing questions *cyclomatic complexity*, *nested block depth* and the two dynamic metrics, developed for this study, are significantly correlated to the student performance and therefore to the observed difficulty of the question (Table 4). Increasing complexity, as defined by increasing values in the four metrics of a tracing question, therefore directly correlates with an increase in difficulty for previously ‘unseen’ code that does not extend beyond the courses content. The definition of ‘unseen’ code is code that is either entirely new code for which the key syntax and language constructs had been taught during the course or a variation on code that had been seen in the context of the course. For example the students may have, as a lab exercise, been asked to write a method that found the highest number in an array of numbers and the ‘unseen’ code might find the lowest number. Therefore it can be argued that the students should have the knowledge required to answer an ‘unseen’ question and that such a question requires them to apply or adapt their existing knowledge in order to solve the question.

Question Type	software metric	Kendall's τ -b (2-tailed)
Tracing	Cyclomatic complexity	0.775*
	Average nested block depth	0.775*
	Number of operands	0.231
	Number of parameters	0.452
	Number of java commands	0.304
	Sum of all operands in the executed statements	0.732*
	Number of commands in the executed statements	0.732*
EiPE	Cyclomatic complexity	0.289
	Average nested block depth	0.109
	Number of operands	0.219
	Number of parameters	-0.040
	Number of commands	0.274
	Number of methods	-0.277

Table 4: Correlations between software metrics and question difficulty [* p < 0.05]

None of the metrics used correlated significantly with the difficulty of the EiPE questions. Although it is possible that questions that require EiPE responses are inherently unsuitable for a metrics approach to predicting difficulty it is just as likely that we have yet to identify metrics capable of performing this task.

6 Conclusion

This research has analysed student responses to two types of exam questions, which are typically used in novice programming exams, code tracing and EiPE. The results have shown that some software metrics, for our dataset, correlate to the difficulty of code tracing exam questions. As a result of this study we suggest that software metrics might be a useful tool in the early prediction of the

difficulty of this type of first year computer programming examination question.

More research is needed into the possible use of software metrics for evaluating EiPE questions and other forms of programming tasks and questions to see whether or not it is possible to develop metrics that are meaningful in those contexts. Further consideration needs to be given to what other metrics may be useful for the analysis of EiPE questions and perhaps to determining the criteria that should be used to determine whether or not any given EiPE question should be included in a set of questions of that type. It is possible that some of the existing metrics could provide useful information if the question set was more homogeneous.

When undertaking this analysis we found aspects of some questions that were not measured by the metrics but that affected the validity of those questions. What the question is assessing may not be what the examiner intended. For example a question that includes mathematical operators or concepts may be testing mathematical knowledge not programming knowledge. Perhaps such questions should be avoided unless the intent is to assess the mathematical concept. Additionally the use of previously ‘seen’ code has the potential to alter the way in which students respond to the question. An EiPE question with relatively complex code may actually be reduced to a simple recall question rather than one that requires an understanding of the code.

In this study we undertook an item discrimination analysis but it appears that some of our questions may have additional issues of validity or of inappropriate item difficulty. It is our recommendation that any future research should include a full item analysis of all questions and include only those questions that have performed adequately in terms of reliability, validity, difficulty and item discrimination in any further analysis. This would reduce the likelihood that any question set contained poorly performing questions that could obscure possible relationships between the data set and software metrics. It could also lead to the development of criteria for each question type that could be used in future to help to ensure that questions meet an appropriate standard and can be meaningfully evaluated using the appropriate software metrics.

Future work will involve applying metrics to other types of questions. This work will include measuring the contribution of each metric to the overall question difficulty with the intention of designing a single weighted metric for each question type. We also intend to verify the findings of this preliminary study firstly with a larger set of examination questions and secondly by designing questions using software metrics as a factor that is considered in that design and evaluating the effectiveness of this approach. Finally, we believe that code writing tasks might also be amenable to the same approach by identifying relevant software metrics and applying them to the model answer and to the student solutions.

7 References

- Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., Rath, J. and Wittrock, M. C. (2001): *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman.
- Biggs, J. B. and Collis, K. F. (1982): *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. New York. Academic Press.
- Clear, T., Whalley, J., Lister, R., Carbone, A., Hu, M., Sheard, J., Simon, B., and Thompson, E. (2008): Reliably Classifying Novice Programmer Exam Responses using the SOLO Taxonomy. *Proc. 21st Annual Conference of the National Advisory Committee on Computing Qualifications (NACCCQ 2008)*, Auckland, New Zealand, 23--30.
- Elliott Tew, A. (2010): Assessing fundamental introductory computing concept knowledge in a language independent manner. PhD dissertation, Georgia Institute of Technology, USA.
- Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D. Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T. L. McGee Thompson, D., Riedesel, C. and Thompson E. (2007): *Developing a computer science-specific learning taxonomy*. SIGCSE Bull. **39**(4): 152-170.
- Halstead, M.H. (1977): *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA, Elsevier Science Inc..
- Kintsch, W. and van Dijk, T.A. (1978): Towards a model of text comprehension and production. *Psychological Review*, **85**, 363-394.
- Klemola, T. (1978): Software comprehension: theory and metrics. Masters Thesis, Concordia University, Montreal, Canada.
- Klemola, T. and Riling, J. (2003): A cognitive complexity metric based on category learning. *Proc. of the 2nd IEEE International Conference on Cognitive Informatics (ICCI'03)*, London, UK, 106 – 112.
- Lister, R., Simon, B., Thompson, E., Whalley, J. and Prasad, C., (2006): Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy, *SIGCSE Bulletin*, **38**(3): 118 - 122.
- Murphy, L., McCauley, R. and Fitzgerald, S. (2012): 'Explain in plain English' questions: implications for teaching. *Proc. of the 43rd ACM technical symposium on Computer Science Education (SIGCSE '12)*, 385-390
- Mathias, K.S., Cross, J.H., Hendrix, T.D., and Barowski, L.A. (1999): The role of software measures and metrics in studies of program comprehension. *Proc. of the 37th Annual Southeast Regional Conference (CD-ROM)*, ACM-SE, **37**, article 13, doi =10.1145/306363.306381
- Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. (2010): Learning Computer Science Concepts with Scratch. *Proc. of the 6th International Computing Education Research Workshop (ICER 2010)*. Aarhus, Denmark, 69-76.
- McCabe, T.J. (1976): A Complexity Measure, *Software Engineering, IEEE Transactions on*, **2**(4), 308- 320.
- Parker, J. R. and Becker, K. (2003): Measuring effectiveness of constructivist and behaviourist assignments in CS102. *Proc. of the 8th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2003)*, Thessaloniki, Greece, 40-44.
- RSA, IBM. http://publib.boulder.ibm.com/infocenter/ieduasst/rtnv1r0/index.jsp?topic=/com.ibm.iea.rsar/plugin_types.html. Last accessed 24 August 2012.
- Sheard, J., Carbone, A., Lister, R. Simon, B. Thompson, E. and Whalley, J. L. (2008): Going SOLO to assess novice programmers, *Proc. of the 13th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITiCSE'08)*, Madrid, Spain, 209-213.
- Shuhidan, S., Hamilton, M. and D'Souza, D. (2009): A taxonomic study of novice programming summative assessment. *Conferences in Research and Practice in Information Technology*, **95**: 147-156.
- Venables, A., Tan, G. and Lister, R. (2009): A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. *Proc. of the 5th International Computing Education Research Workshop (ICER 2009)*, Berkeley, CA, USA, 117-128. Berkeley, California, August 10-11, 2009. pp. 117-128.
- Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P. and Prasad, C. (2006): An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Australian Computer Science Communications*, **52**: 243-252.
- Whalley, J., Clear, T., Robbins, P., and Thompson, E. (2011): Salient Elements in Novice Solutions to Code Writing Problems. *Conferences in Research and Practice in Information Technology*, **114**: pp. 37-46.

Appendix

Example of a typical EiPE question and a typical code tracing question:

Question 5A

In plain English, explain the purpose of this method. Note that more marks will be gained by correctly explaining the purpose of the code than by giving a description of what each line does.

```
public int method(int x, int y)
{
    int result =x;
    if(x < y)
    {
        result = y;
    }
    return result;
}
```

Question 1C

Complete the trace table below to show what happens when this method is executed with the parameter *limit* equal to 4.

```
public int method(int limit)
{
    int result = 0;
    for(int i = 0; i<= limit; i++)
    {
        result += 2;
    }

    return result;
}
```

i	result
0	0

| Initialisation

Revisiting models of human conceptualisation in the context of a programming examination

Jacqueline Whalley and Nadia Kasto

School of Computing and Mathematical Sciences

AUT University

PO Box 92006, Auckland 1142, New Zealand

{nkasto, jwhalley}@aut.ac.nz

Abstract

This paper reports on an evaluation of the Block model for the measurement of code comprehension questions in a first semester programming examination. A set of exam questions is classified using the Block model and two commonly employed taxonomies, SOLO and Bloom. We found that some of the problems inherent in the application of Bloom and SOLO taxonomies also exist in the Block model. Some of the difficulties associated with SOLO and Bloom’s taxonomy are due to the wide breadth of the dimensions. These difficulties are to some degree mitigated by the limited breadth of the Block model dimensions and we found that the Block model provided a better way of describing novice programming code comprehension tasks because of the increased granularity that it provides.

Keywords: code comprehension, novice programmers, Block model, SOLO, Bloom’s taxonomy.

1 Introduction

Teachers of computer programming have experienced difficulty in judging the cognitive complexity of learning tasks and test items. A relatively accurate and simple way is required for determining the difficulty inherent in our teaching and assessment programs: “...we as educators are continually underestimating the difficulty of the tasks that we are asking students to undertake” (Whalley, Clear and Lister 2007).

Computer science educators have attempted to apply models and taxonomies of human conceptualisation to aspects of the teaching and learning of computer programming with varying degrees of success. The most widely adopted taxonomies to date have been the Bloom (Bloom 1956) and SOLO (Biggs and Collis 1982) taxonomies. Recently a new taxonomy has been developed specifically for application to the design of tasks for computer programming. This paper reports on an investigation of the use of that model to determine the difficulty of a set of test questions.

2 Background

In 1956, Bloom produced a taxonomy that consisted of a hierarchy of learning objectives ranked according to their

expected cognitive complexity (Figure 1). The taxonomy is a behavioural classification system of educational objectives. Many variants of the taxonomy have been proposed but the most widely accepted (Figure 1) is the revised Bloom’s taxonomy (Anderson et al. 2001). This version of the taxonomy adds a knowledge dimension, which specifies the type of information that is processed, to a revised version of the original cognitive process dimension. Traditionally a strict inclusive hierarchy has been assumed for the cognitive process dimension where each category is assumed to include lower ones.

Bloom	Revised Bloom
Evaluation	Create
Synthesis	Evaluate
Analysis	Analyse
Application	Apply
Comprehension	Understand
Knowledge	Remember

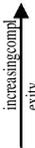


Figure 1: The cognitive process dimension; (left) Bloom’s and, (right) revised Bloom’s taxonomy

Bloom’s taxonomy has been applied to computer science for course design and evaluation (Scott 2003), structuring assessments (Lister and Leaney 2003, Lister 2001), specifying learning outcomes (Starr, Manaris and Stavely 2008) and comparing the cognitive difficulty of computer science courses (Oliver et al. 2004).

The revised and the original Bloom’s taxonomy have been used in attempts to improve the instruction and assessment of programming courses (e.g., Abran et al. 2004, Shneider and Gladkikh, 2006, Thompson et al. 2008, Khairuddin and Hashim 2008, Alaoutinen and Smolander 2010, Whalley et al. 2006, Whalley et al. 2007, Shuhidan, Hamilton and D’Souza 2009).

The use and interpretation of Bloom and the revised Bloom’s taxonomy for describing computer science tasks has been found to be problematic (Fuller et al. 2007, Thompson et al. 2008, Shuhidan, Hamilton and D’Souza 2009, Meerbaum-Salant, Armoni and Ben-Ari 2010). Much of the research shows that it can be difficult to reach a consensus on an interpretation for the computer programming education domain (Johnson and Fuller 2006). In a recent study Gluga et al. (2012) confirmed that many of the ambiguities in the application of Bloom’s taxonomy to the assessment of computer programming are due to the necessity to have a deep understanding of the learning context in order to achieve an accurate classification. They also noted that the classifiers often had preconceived misunderstandings of the categories and differing views on the complexity of

Copyright © 2013, Australian Computer Society, Inc. This paper appeared at the 15th Australasian Computer Education Conference (ACE 2013), Adelaide, South Australia, January-February 2013. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 136. A. Carbone and J. Whalley, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

tasks and the sophistication of the cognitive processes required to solve them. This may be due to the difficulty that the educators have remembering the cognitive complexity of such a task when they were learning to program. A much higher cognitive load exists for a novice programmer writing a simple function than for an experienced programmer. Additionally it has been reported that the ordering of cognitive tasks in Bloom’s taxonomy does not readily map to the learning trajectories of many novice programmers (Lahtinen 2007).

As a result of these difficulties, several variants of Bloom’s taxonomy have been proposed specifically for computer programming education (e.g., Schneider and Gladkikh 2006, Fuller et al. 2007, Bower 2008). These variants have not been widely adopted by computer science educators and researchers. Perhaps this is partially due to the fact that the appropriateness of Bloom’s taxonomy for the design of learning activities and assessments has been disputed. The presupposition that there is a necessary relationship between the questions asked and the responses elicited is not a valid one because a question could potentially elicit responses at different levels (Hattie and Purdie 1998).

Biggs and Collis (1982) surmised that Bloom levels reflect a teacher imposed view of what it means to have achieved full mastery whereas SOLO levels come from an understanding of the student learning process. The focus of Bloom is to assist in the development of educational objectives, while the SOLO taxonomy focuses on the cognitive process used to solve problems. SOLO, unlike Bloom, does not assume a relationship between the task and the outcome so outcomes to a specific task may be at different levels for different students. Additionally, while Bloom separates knowledge from the intellectual processes that operate on this 'knowledge', the SOLO taxonomy is primarily based on the processes of understanding used by the students when solving problems. Therefore, knowledge is inferred in all levels of the SOLO taxonomy. It may be due to these differences that educators and researchers have had greater success in using SOLO to describe programming tasks (code comprehension and code writing), to classify student responses to those tasks and to gain some insight into the students’ cognitive processes (e.g., Lister et al., 2006, Philpott, Robbins and Whalley 2007, Sheard et al. 2008, Clear et al. 2008).

Both taxonomies have been used, independently, to analyse the same set of programming assessment questions and responses (Whalley et al. 2006). Thompson et al. (2008) noted that the Bloom category for a programming task can be meaningfully mapped to a number of categories in the SOLO taxonomy and that a combined version of these taxonomies may provide a richer model with which to design and describe programming tasks. Inspired by Thompson’s observation a hybrid taxonomy was proposed that combines aspects of the Bloom and SOLO taxonomies (Meerbaum-Salant, Armoni and Ben-Ari 2010). The combined taxonomy

consists of the SOLO categories of *unistructural*, *multistructural* and *relational* and three Bloom categories *understand* (U), *apply* (A) and *create* (C). The taxonomy was structured so that the three SOLO levels formed super-categories each containing the three Bloom levels as subcategories (Figure 2) . This taxonomy was then used to analyse the correlation between student performance on a task and the relative complexity of the task as defined by the classification of the task using the combined taxonomy. The authors believe that their “findings suggest that the combined taxonomy captures the cognitive characteristics of CS practice”. They also recommend this integration of taxonomies as a research framework that is applicable to the specific needs of CS education research. However, they also note that the taxonomy requires further investigation and validation. To date this work has not been reported in the literature.

Unistructural	Multistructural	Relational
U A C	A A C	U A C

Figure 2: The combined taxonomy

The Block model (Schulte 2008) is an educational model of program comprehension. It is structured as a table consisting of three knowledge dimensions and four hierarchical levels of comprehension. The table consists of 12 blocks (cells) and each block is designed to highlight one aspect of the program comprehension process (Figure 3). The conceptualisation of the hierarchical levels takes inspiration from Kintsch’s expanded text comprehension theory (1998).

Macro structure	Understanding the overall structure of the program text	Understanding the “algorithm” of the program	Understanding the goal / the purpose of the program in its context
Relations	References between blocks (e.g.: method calls, object creation, accessing data)	Sequence of method calls	Understanding how subgoals are related to goals, how function is achieved by subfunctions
Blocks	Regions of interest (ROI) that syntactically or semantically build a unit	Operation of a Block, a method or ROI (as a sequence of statements)	Function of block, maybe seen as a subgoal.
Atoms	Language elements	Operation of a statement	Function of a statement. For which goal is only understandable in context
	Text surface	Program execution (data flow and control flow)	Functions (as means or as purpose, goals of the program)
	Structure		Function

Figure 3: The Block model (Schulte 2008)

Question	Type	Revised Bloom's		Block Model		SOLO	% correct answers
			cognitive dimension	level	comprehension dimension		
2	Basics		Remember	Atom	Text surface	U	50%
4	Syntactic errors	A	Remember	Atom	Text surface	U	68%
		B	Understand	Block	Text surface	M	53%
		C	Understand	Relations	Text surface	M	31%
7A	Tracing		Apply	Block	Execution	M	77%
7B	Tracing		Apply	Block	Execution	M	64%
7C	Tracing (with selection)		Apply	Block	Execution	M	81%
7D	Tracing (with iteration)		Apply	Relations	Execution	M	22%
7E	Tracing (with iteration)		Apply	Relations	Execution	M	27%
5	Skeleton Code		Analyse	Relations	Functions	R	42%
6	Parsons Puzzle		Apply	Block	Functions	M	60%
10A	Code Intent		Understand	Macro	Functions	R	36%
10B	Code Intent		Understand	Macro	Functions	R	9%
10C	Code Intent		Understand	Macro	Functions	R	6%

Table 1: Classification of exam questions

The intention behind the Block model's development was to provide a relatively simple model, compared with other existing taxonomies and models, to support research into the teaching of computer programming. The model was evaluated as a tool for the planning and evaluation of lessons about algorithm design (Schulte 2008). It was found that the Block model was simple, constructive and communicative. However the model has not yet been used as a framework for research into the teaching and learning of computer programming.

In a recent paper the Block model was used to map a variety of selected models of program comprehension in order to assist in the conceptualisation of those models (Schulte et al. 2010). As a result of this comparative analysis of models the authors suggest that the process of knowledge acquisition by novice programmers described in terms of the Block model might be represented as a holey patchwork quilt and that the Block model might help us identify what holes (empty cells) exist and why a student's knowledge is "fragile". We were interested in investigating the usefulness of the Block model for measuring and evaluating programming tasks and also for investigating the cognitive processes employed by students to solve the problems.

In this preliminary study we employed a set of programming comprehension questions, from a first semester programming examination, in order to analyse the similarities and differences of the Block model with other models.

3 Analysis and Discussion

What follows is a discussion of the analysis of a small set of program comprehension questions, given in the same pen and paper examination, collated by question type. Table 1 gives an overview of the classification of the questions. The revised Bloom classification was carried out using the vignettes and principles described by Thompson et al. (2008) and Whalley et al. (2006). In accordance with this set of guidelines we classified the cognitive process dimension at the category level rather than the sub category level. In classifying the questions using the SOLO taxonomy we applied the principles and

guidelines provided by Biggs and Collis (1982) and from the SOLO categories established by the BRACElet project for 'code intent' comprehension tasks (Clear et al. 2008). The Block model classification was carried out using the cell descriptions shown in Figure 3.

One challenge we faced was in determining exactly what an atom or a block is. It could be argued that this is dependent on the stage of development that the individual learner has reached. This observation has been previously made with respect to salient elements in novice programming tasks (Whalley et al., 2010). Here in assigning classifications we have assumed a norm for all students based on our experiences in teaching novice programmers. We have taken the notion of an *atom* to be the simplest salient element (for example a variable declaration and assignment) and a *block* to be a single method, loop or selection statement. Therefore at the *relations* level we consider a relationship to be a reference between blocks or between a block and an atom.

For each question the student performance on that question was also analysed. In our analysis we are interested in what skills, knowledge base and cognitive processes are required to successfully answer the question. Finally we then compared the actual relative difficulty of the questions in the context of the examination (as indicated by the percentage of students who gave a fully correct answer) with the levels of difficulty of those questions as indicated by the taxonomies and models.

Question 2: Matching Terms to Code

Question 2 presented students with a class definition that had ten lines of code underlined and each annotated with a letter. Students were asked to match 7 definitions to the appropriate line of code. This question required students to recall factual knowledge and was classified as *remember*. In terms of SOLO this question requires students to focus on a single language construct and is therefore a *unistructural* question.

Because students are focusing on a single language element this question is considered to be at the atom level

in the Block model. The *text surface* dimension of the Block model is associated with the external representation of the program, “*it is the code a person reads in order to comprehend the program*” (Schulte et al. 2010). In order to answer question 2 the students do not need to go beyond understanding the rules of discourse (grammar) of the program code. They certainly do not need to understand or have knowledge of the data and control flow or goal of the atom of code in order to answer this question correctly. Therefore this question is an *atom* level, *text surface* question.

Question 4: Syntax Errors

In question 4 (see Appendix) students were asked to find 8 of 11 syntax errors in a complete class. The type of syntax error had a great affect on the number of students who were able to correctly locate and identify the error. It seems, not unexpectedly, that the difficulty of the task (as measured by student performance on the task) is related to the type of knowledge that is required which in turn is directly related to the type of bug or error to be identified. We found that when we mapped each error identification question to the Block model clear groupings emerged based on the level of comprehension required to reach the correct answer. The lowest level of syntactic errors, which we grouped together as 4A, consisted of errors such as missing semicolons, a missing bracket in a method declaration and typographical errors such as Return rather than the correct return keyword. All of these errors can be found without reference to the rest of the program structure. They focus on a language element and therefore with respect to the level of student program comprehension required to answer the question they can be classified as *text surface* at the *atom* level. Identifying these types of errors can also be considered to be a *unistructural* task and in terms of Bloom they require the students to *recognise* an error that they would have seen repeatedly during their course of study.

The syntactic errors that we grouped as question 4B consist of mismatches either between the return data type of a method and the data type of the value returned or between a parameter identifier in the method declaration and the identifier used to represent that method parameter in the method body. These errors are all located within a block of code and consist of a sequence of atoms. One error was positioned in a selection statement. In order to locate these errors the students must understand the syntactic structure of the block so these error identification tasks were classified as requiring *text surface* knowledge at the *block* level. It is not necessary to operate at the *relations* level in order to identify these errors. We classified the 4B errors to the SOLO *multistructural* category because they focus on more than one language construct but to answer correctly they do not require the students to understand the relationship between the constructs and the problem can be solved by knowing the required structure of the code rather than the purpose or goal of the code. In order to identify this type of error students must not only recall basic syntax rules but also identify where there is an incorrect application of the rule. In order to do this the students must *understand* (Bloom’s category) the rule.

Finally, syntactic errors that were grouped together as 4C consisted of bugs such as an incorrect method call or a data type mismatch for a global variable. In order to recognize this type of error, the students need to be aware of the relationships between various blocks in the code and therefore required comprehension of *relations*. In order to identify these bugs the students are still operating at the *text surface* where an understanding does not need to extend beyond the application of their knowledge of the ‘grammar’ of the code.

While the different forms of question (4A, 4B, 4C) were classified into three separate categories in the Block model they were classified into only two different categories when Bloom and SOLO classifications were applied. The Block model was the only classification system to put the three different types of questions into separate categories.

Question 7: Code Tracing

Tracing questions are solved by tracking data through the code line by line. This question type has not been previously classified using the SOLO taxonomy. However, in a study that analysed student answers to ‘code intent’ questions it was noted that “*a student may hand execute code and arrive at a ... [correct]... final value but ... the student may not manifest an understanding of what the code does*”. Such student responses were classified as *multistructural* (Lister et al. 2006). Extrapolating this to tracing questions it is clear that it is not necessary to understand the purpose of the code to reach the correct answer and question 7 A-E should be classified as *multistructural*.

These questions require the students to apply a known process or strategy and are therefore classified as *apply* in the revised Bloom’s cognitive dimension.

The students need to have knowledge of the data flow and in some cases control flow of a simple Java method in order to answer code tracing questions. In order to operate at the *program execution* level they must also operate at the lower *text surface* level. They do not need to extend to the *functions* domain of the Block model. Therefore, the tracing questions in this exam are all posed within the *program execution* knowledge dimension of the Block model.

The aspect in which these questions differ is the level of the task when classifying the questions using the Block model. The Block model was the only system that differentiated amongst these tracing questions. The questions were classified into two different levels within the Block model. For these questions code that contained iteration were classified at the *relations* level whereas code without iteration were classified as *block* level questions.

Question 5: Skeleton Code (with scaffolding)

The skeleton code for question 5 is a class definition, taking up a page and a half, containing two private data members (one of which is an ArrayList), a single constructor, and three methods, which add to, delete from, and print the contents of the ArrayList. After the code, the students are set the following task for refactoring the code: “The table below shows the missing lines of code, but not necessarily in the correct order. It also has one extra line of code that is not needed. Identify

which line of code should go where ...” In-line comments are provided as a scaffold to help the students identify the appropriate lines of code. The scaffolding means that it is not necessary for the students to identify the overall goal of the missing lines and the blocks in which the line must be placed because this is provided. It does still, however, require them to understand the various relationships between lines of code in order for them to select the correct missing line. For example see Figure 4 where there is a need to understand the connections between fields as parameters to an external constructor method for a Lot object and the Lot object and an ArrayList method call as well as the sub-goals of these method calls. Question 5 was therefore classified as a SOLO *relational* question and in the *relations-functions* of the Block model. This question requires the students to differentiate the relevant from the irrelevant lines of code and to focus on the sections of code within the class that are relevant to the differentiation task. Therefore this question was classified as *analyse*. A similar skeleton code question has been reported previously and was also classified at the *analyse* level of Blooms cognitive process dimension (Whalley et al. 2006).

```
private ArrayList<Lot> lstLots;
private int nextLotNumber;
...
/**
 * A simple model of an auction
 * @author David J. Barnes and Michael
 * Kolling */
public void enterLot(String description)
{
    //create new Lot
    Lot lot = new Lot(nextLotNum,description);
    //store it in the ArrayList
    <missing code>
    nextLotNumber++;
}
```

Figure 4: Part of question 5 code (adapted from Barnes and Kölling 2006)

Question 6: Parsons puzzle (with structure)

Question 6 is a Parsons puzzle (Parsons and Haden 2006) where students are presented with jumbled lines of code for a Java method (see Appendix). They are provided with the purpose of the method which is to count the number of occurrences of a character in a string and a structure for the method defined by a set of nested braces and blank lines. The students are required to place the lines of code into the correct order.

Classifying this question using SOLO is difficult. It could be argued that even though the students are provided with the overall purpose of the code they still have to understand the code as a whole in order to reach the correct answer. Therefore it should be considered to be a *relational* question. Additionally if we take this viewpoint the revised Bloom’s cognitive level of the question must be *analyse* because the students are determining how the lines of code fit within the overall structure and purpose of the code. However research using these puzzles points to the fact that students typically *apply* a set of heuristics to solve the problem (Denny et al. 2008). For example, the final line of the method must be the return statement and the first line the method header. Even in determining the position of the

loop in relation to the selection statement the variable *i* is defined in the loop and then used in the ‘if statement’. Understanding the relationship between the two lines of code and the variable *i* can be seen as applying a more sophisticated heuristic. On the other hand it could be seen as manifesting an understanding of the purpose of the variable *i*. Either way in terms of SOLO, the question is *multistructural* because although connections between parts of the code must be made, the question does not require meta-connections to be made.

The Parsons puzzle examined here is classified at the *apply* level, in the revised Bloom’s cognitive dimension, because it is possible to solve this problem correctly by applying known heuristics.

This question requires students to operate at the *block* level. It is not necessary for them to be able to understand the connections between the blocks to solve the problem because they can use heuristics. Although the students are given the overall goal of the method it is possible to solve this Parsons puzzle without understanding the overall goal. However it seems that the students must at least understand the sub-goals of the constituent blocks and atoms in order to solve this puzzle correctly. Therefore we have classified this puzzle in the *functions* knowledge domain. It should be noted that a more complex puzzle, without scaffolding, may require a deeper understanding of the logic and flow of the algorithm and be at the *relations* level of the *functions* domain. Therefore, unlike tracing questions, we cannot claim that all Parsons puzzles have a predetermined classification.

Question 10: Code Intent

Questions 10 A, B and C all required the students to explain the purpose or goal of a single method. It is clear that such a question moves beyond the structure of the code, data flow and control flow, and is within the Function cognitive dimension of the Block model. As an example we will consider question 10B (Figure 5). To solve this question the students need to understand the connections between the three blocks of code in order to infer an overall purpose.

```
public void method10B(int iNum)
{
    for(int iX = 0; iX < iNum; iX++) Block 2
    {
        for(int iY = 0; iY < iNum; iY++)
        {
            System.out.print("*"); Block 3
        }
    }
    System.out.println();
}
```

Figure 5: Question 10B and its constituent blocks

There are a number of possible ways that a student might solve this problem. They might apply a top-down comprehension strategy by identifying the sub-goal of *Block 3* before trying to understand the function of *Block 2*. When considering *Block 2* they see *Block 3* as an atom (which prints a line of *iNum* stars) and that *Block 2* executes *Block 3* followed by a carriage return *iNum* times. The outer block might be processed in a similar way in order to arrive at the purpose of the method. In this way it can be argued that a student is reaching an

understanding of the relationship between the three blocks and then inferring an overall purpose. The same outcome might be achieved by tracing the code, a bottom-up strategy, in order to try to see the relationships between the blocks and arrive at a conclusion as to the purpose of the code. Or they may apply a combination of both. Regardless of the strategy they apply, to reach a correct answer the student is operating at the highest level within the *functions* dimension because an “understanding the goal or function of the program” is required. ‘Code intent’ questions, similar to the ones in this examination have been consistently classified as SOLO *relational* (see Clear et. al 2008) and this gives weight to our classification of questions 10A-C as *relational*. In past work ‘code intent’ questions have been classified as *understand* (Thompson et al. 2008, Whalley et al. 2006). Our initial classification was at this level. However we believe that the cognitive processes used by novice programmers when trying to solve ‘code intent’ questions are more complex than previously assumed. A fuller discussion of this aspect of using Blooms taxonomy to classify program comprehension tasks is provided in the next section.

3.1 Using Bloom’s taxonomy

Like many educators in science disciplines we have found it difficult to apply the Bloom and the revised Bloom taxonomies. There have been several studies that indicate that the order of the levels changes depending on the task. For example in a test on atomic structure it was found that *synthesis* and *evaluation* were placed between *knowledge* and *comprehension*. A test related to glaciers found that *synthesis* lay between *knowledge* and *comprehension* (Kropp and Stocker 1966). Similarly, we believe that the cognitive dimension hierarchy does not map comfortably with computer programming tasks.

In classifying ‘code intent’ questions Thompson et al.’s (2008) revised Bloom vignettes and definitions indicate that this type of question is at the *understand* level. In the revised Bloom’s taxonomy *understand* is defined as ‘*constructing meaning from instructional messages*’ which is interpreted by Thompson et al. (2008) to include “*explaining a concept or an algorithm or design pattern*”. Tracing questions were classified, by Thompson et al. (2008) at the higher revised Bloom level of *apply*. *Apply* is defined as “*carrying out or using a procedure in a given situation*” and clearly hand execution of code is a process which students must apply in order to answer a code tracing question.

Past research has shown us that novice programmers find ‘code intent’ questions more difficult than tracing questions and Parsons puzzles (Lopez et al. 2008). A study which examined the approaches of experts vs. novices in solving these types of problems also illustrated that there is likely to be a higher cognitive load and more complex cognitive processes involved in solving a previously unseen ‘code intent’ question than for an unseen tracing question (Lister et al. 2006). Additionally they found that even experts sometimes approach ‘code intent’ questions by first partially tracing the code in order to discover the code’s purpose. In classifying questions to Bloom the highest cognitive process level necessary to solve the problem should be used.

Consequently, at the lowest possible level ‘code intent’ questions must be *apply*. We believe that code intent tasks are more complex than has previously been assumed. It is likely that students first break down the code into manageable chunks and then try to determine the goal of each chunk, possibly by using a tracing strategy. At this point it is likely that they try to start mapping this code to their existing knowledge. Subsequently the students try to establish how the parts relate to one another and attempt to arrive at an overall purpose for the code. If this viewpoint is accepted then it is evident that ‘code intent’ questions require the students to be thinking at the *analyse* level. This classification would be more in line with the SOLO and Block classifications for ‘code intent’ questions and would better reflect the level of difficulty of such questions for novice programmers.

3.2 Reflections on the Block model

The Block model classification of this small set of exam questions seems to indicate that there is a relationship between the Block classification of a question and the observed difficulty of a question.

The average % of fully correct answers for all questions classified into a block for each block in the Block model is shown in Figure 6. When compared with SOLO and Bloom (see Table 1) the Block model classification levels appear to more accurately match the relative difficulties of code comprehension tasks for novice programmers.

Macro structure			17%
Relations	31%	24.5%	42%
Blocks	53%	74%	60%
Atoms	59%		
	Text surface	Execution	Functions

Figure 6: Average % fully correct answers

This relationship is particularly evident when examining the results by question type. For example the tracing questions (question 7A – E) become progressively more difficult for the students to answer as the block level and knowledge dimensions increase (see Table 1 and Figure 6). However the teaching context of the knowledge required to successfully solve a question affects the difficulty of that question. The students found 7C was much easier (81% correct answers) than question 7B (64% correct). On closer examination question 7C required students to determine if a number was outside of a given range. The selection statement used a logical or. This code had been covered in detail in class using a “range doodle” (Whalley et al. 2007). Many of the scripts had such doodles on them indicating that although the code was presented as the opposite logic of the class room example, which checked if values were within a range, the teaching had an impact on the learning of the students. Question 7B on the other hand was a simple remainder operation. The fact that 36% of students could not solve this simple problem as well as they could 7C suggests that the students lack basic mathematical knowledge that was assumed in the teaching of

programming for this cohort. Despite these differences overall tracing problems which are *program execution* knowledge domain questions that were posed at the *block* level were easier than those posed at the *relations* level.

If we map the SOLO classification of our questions to the Block model classification a pattern emerges that shows a possible relationship between Block model levels and SOLO (Figure 7).

Macro structure			Relational
Relations	Multistructural	Multistructural	Multistructural
Blocks	Multistructural	Multistructural	Multistructural
Atoms	Unistructural		
	Text surface	Execution	Functions

Figure 7: Mapping of SOLO & Block model classifications

A relationship had been hypothesised by Schulte et al. (2010) and while our findings support a mapping we propose that the *relations* level actually maps to the SOLO *multistructural* level and not the *relational* (Figure 8). We found in our exam that questions at the relations level across all three knowledge dimensions were at the SOLO level of *multistructural*. It is important to note the distinction between relations (references between blocks) and ‘thinking’ at a *relational* level when classifying exam questions using the Block model.

Block model	SOLO (Schulte et al. 2008)	SOLO (revised mapping)
Macro	Relational	Relational
Relations	Relational	Multistructural
Block	Multistructural	Multistructural
Atom	Unistructural	Unistructural

Table 2: Mapping the Block model to SOLO

Figure 8 shows the mapping between the Bloom and Block model classifications. As observed for SOLO there is a general trend of difficulty as you progress up the Block levels and this was also reflected in decreasing student achievement.

There also appears to be a general trend of increasing cognitive complexity required to solve the questions as you move from text structure to functions across the Block model knowledge dimensions. However, this trend is not present in the student performance data on the set of questions reported in this paper. It is possible that this trend was not observed because we do not have sufficient data for some of the blocks. For some questions it was difficult to determine which block the question should be classified to if the question lay on the boundary. It may be necessary to further define the blocks and provide vignettes to guide the classification process.

Macro structure			Analyse
Relations	Understand	Apply	Analyse
Blocks	Understand	Apply	Apply
Atoms	Remember		
	Text surface	Execution	Functions

Figure 8: Mapping Bloom & Block model classifications

4 Conclusion

It is important to note that many of the limitations that exist for the use of Bloom and SOLO also exist for the Block model. In particular it is necessary to understand the context of learning and what prior exposure students have to the information required. In under taking this research we have noted that when educators attempt to design “better models” they somehow end up with models that appear to be revisions of existing taxonomies. In this case it appears that the Block model might actually be a hybrid of a revised SOLO and a revised Bloom’s taxonomy.

Based on our experience SOLO still seems to be the most straightforward model to apply but in using SOLO we lose the granularity to examine programming exam questions because those tasks are largely *multistructural*. A recent survey of first year programming exams found that 20% of questions in CS1 courses were tracing questions and 9% were explain questions (Simon et al. 2012). The main advantage of the Block model is that it provides us with a way of describing these novice programming tasks that gives us a level of granularity which allows us to distinguish between similar tasks in a way that SOLO or Bloom’s taxonomy cannot.

The mapping of tasks to the Block model reveals ‘holes’ in the coverage of our examination of code comprehension. We do not have questions that are about the execution and functions of atoms or questions that require text surface and execution knowledge at the macro structure level. Examinations reflect the focus of our teaching. The lack of coverage of the Block model leads us to question whether or not we have it right. Could we be missing key tasks that might enable student learning? If we do cover the entire Block model can we improve code comprehension? Perhaps an increased focus on these missing areas during instruction will help students to develop advanced understanding more rapidly.

The work reported here is a preliminary look at the usefulness of the Block model for measuring and evaluating programming tasks and also for investigating the cognitive processes employed by students to solve the problems. In order to explore this further we intend to analyse a larger set of examination questions. We also plan to use the Block model to design assessment tasks and to attempt to establish the level at which the students are actually operating by using think-out-loud interviews.

In our analysis we have omitted code writing tasks, largely because the model was originally designed for comprehension tasks. But it would be interesting to revisit the Block model with a focus on code writing tasks. We believe that the Block model, with minor refinements, might also provide a useful framework for research and teaching of code writing tasks.

5 References

- Abran, A., Moore, J., Bourque, P., DuPuis, R. and Tripp, L. (2004): Guide to the Software Engineering Body of Knowledge - 2004 Version SWEBOK®, Los Alamitos, CA, IEEE-CS - Professional Practices Committee.
- Alaoutinen, S. and Smolander, K. (2010): Student Self-Assessment in a Programming Course Using Bloom’s

- Revised Taxonomy. *Proc. of the 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10)*, 155–159. ACM Press.
- Anderson, L. W., Krathwohl, D. R., 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*. Longman.
- Barnes, D.J. and Kolling, M. (2006): *Objects First with Java: A Practical Introduction using BlueJ (3rd Edition)*. England, Pearson Education Ltd.
- Biggs, J. B. and Collis, K. F. (1982): *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. New York. Academic Press.
- Bloom, B. S. (1956): *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*. Addison Wesley.
- Bower, M. (2008): A Taxonomy of Task Types in Computing. *SIGCSE Bulletin*, **40**(3): 281–285.
- Clear, T., Whalley, J., Lister, R., Carbone, A., Hu, M., Sheard, J., et al. (2008): Reliably Classifying Novice Programmer Exam Results using the SOLO Taxonomy. *Proc. of the 21st Annual NACCQ Conference*, Auckland, New Zealand, 23-30.
- Denny, P., Luxton-Reilly, A. and Simon, B. (2008): Evaluating a New Exam Question: Parsons Problems. *Proc. of the 2008 International Workshop on Computing Education Research (ICER '08)*, Sydney, Australia, 113-124.
- Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D. Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T. L. McGee Thompson D., Riedesel, C. and Thompson E. (2007): *Developing a computer science-specific learning taxonomy*. *SIGCSE Bull.* **39**(4): 152-170.
- Gluga, R., Kay, J., Lister, R., Kleitman, S. and Lever, T. (2012): Overconfidence and confusion in using Bloom for programming fundamentals assessment. *Proc. of the 43rd ACM technical symposium on Computer Science Education (SIGCSE '12)*, 147–152: ACM Press.
- Hattie, J. and Purdie, N. (1998): *The SOLO model: Addressing fundamental measurement issues*. In B. Dart & G. Boulton-Lewis, (Eds.), *Teaching and Learning in Higher Education*, 145–176. ACER Press.
- Johnson, C. G. and Fuller, U. (2006): Is Bloom's taxonomy appropriate for computer science. In A. Berglund (Ed.), *Proc. of the 6th Baltic Sea Conference on Computing Education Research (Koli Calling 2006)*, Koli National Park, Finland, 120-123.
- Khairuddin, N. N. and Hashim, K. (2008): Application of Bloom's taxonomy in software engineering assessments. *Proc. of the 8th conference on Applied computer science (ACS'08)*, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 66-69.
- Kintsch, W. (1998): *Comprehension: a paradigm for cognition*. Cambridge University Press.
- Kropp, R. P. and Stroker, H. W. (1966): *The construction and validation of tests of the cognitive processes as described in the taxonomy of educational objectives*. Florida State University, Institute of Human Learning and Department of Educational Research and Testing.
- Lahtinen, E. A. (2007): Categorization of Novice Programmers: A Cluster Analysis Study. *Proc. of the 19th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, 32-41. Joensuu, Finland.
- Lister, R. (2001): Objectives and Objective Assessment in CS1. *Proc. of the thirty-second SIGCSE technical symposium on Computer Science Education (SIGCSE '01)*, 292-296: ACM Press.
- Lister, R. and Leaney, J. (2003): First Year Programming: Let All the Flowers Bloom. *Proc. of the 5th Australasian Computing Education Conference (ACE2003)*, Adelaide, Australia, 221-230.
- Lister, R., Simon, B., Thompson, E., Whalley, J.L. and Prasad, C. (2006): Not seeing the forest for the trees: novice programmers and the SOLO taxonomy, *Proc. of the 11th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITiCSE '06)*, Bologna, Italy, 118-122.
- Lopez, M., Whalley, J., Robbins, P. et al., (2008): Relationships between reading, tracing and writing skills in introductory programming. *Proc. of the 4th International Computing Education Research Workshop (ICER 2008)*. Sydney, Australia, 101-112.
- Oliver, D., Dobebe, T., Greber, M. and Roberts, T. (2004): This course has a Bloom Rating of 3.9. *Proc of the 6th Australasian Computing Education Conference*, Dunedin, New Zealand, 227-231,
- Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2010): Learning Computer Science Concepts with Scratch. *Proc. of the 6th International Computing Education Research Workshop (ICER 2010)*. Aarhus, Denmark, 69-76.
- Parsons, D. and Haden, P. (2006): Parson's programming puzzles: a fun and effective learning tool for first programming courses. *Proc. of the 8th Australian conference on Computing Education, Darlinghurst, Australia*, 157–163.
- Schulte, C., Busjahn, T., Clear, T., Paterson, J. and Taherkhani, A. (2010): An introduction to program comprehension for computer science educators. *Proc. of the 2010 ITiCSE Working group reports (ITiCSE-WGR'10)*, Ankara, Turkey, 65-86.
- Schulte, C. (2008): Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching. *Proc. of the 4th International Workshop on Computing Education Research (ICER 2008)*, Sydney, Australia, 149-160.
- Scott, T. (2003): Bloom's taxonomy applied to testing in computer science classes. *Journal of Computing in Small Colleges*, **19**(1): 267-274.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E. and Whalley, J. L. (2008): Going SOLO to assess novice programmers, *Proc. of the 13th annual SIGCSE conference on Innovation and Technology in Computer Science Education (ITiCSE'08)*, Madrid, Spain, 209-213.

Shuhidan, S., Hamilton, M. and D'Souza, D. (2009): A taxonomic study of novice programming summative assessment. *Conferences in Research and Practice in Information Technology*, 95: 147-156.

Simon, Sheard, J., Carbone, A., Chinn, D., Laakso, M., Clear, T., de Raadt, M., D'Souza, D., Lister, R., Philpott, A., Skene, J. and Warburton, G. (2012): Introductory programming: examining the exams. *Proc. of the 14th Australasian Computing Education Conference (ACE2012)*, Melbourne, Australia, 61-70.

Starr, C. W., Manaris, B. and Stalvey, R. H. (2008): *Bloom's Taxonomy Revisited: Specifying Assessable Learning Objectives in Computer Science*. SIGCSE Bulletin, 40(1): 261-265.

Thompson, E., Luxton-Reilly, A., Whalley, J., Hu, M. and Robbins, P. (2008): Bloom's Taxonomy for CS assessment. *Proc. 10th Australasian conference on Computing Education (ACE 2008)*, Wollongong, NSW, Australia, 155-162.

Whalley, J., Clear, T. and Lister, R. (2007): *The many ways of the BRACElet project*. Bulletin of Applied Computing and Information Technology, 5(1). Retrieved August 3, 2012 from http://www.naccq.ac.nz/bacit/0501/2007Whalley_BRACELET_Ways.htm

Whalley, J., Prasad, C. and Kumar, P. K. A. (2007): Decoding doodles: novice programmers and their annotations, *Proc. of the 9th Australasian conference on Computing Education*, Ballarat, Victoria, Australia, 171-178.

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A. and Prasad, C. (2006). An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Proc. of the 8th Australasian Computing Education Conference (ACE2006)*, Hobart, Australia, 243-252.

Whalley, J., Clear, T., Robbins, P., and Thompson, E. (2011): Salient Elements in Novice Solutions to Code Writing Problems. *Conferences in Research and Practice in Information Technology*, 114: 37-46.

Appendix

Question 4

```
import java.util.ArrayList;
public SimpleShop{

private String sName; A - missing ;
private String sPhoneNumber;
private String aAddress;
private ArrayList lstInventory;
private double dTotalAmountSold; A - missing )

public SimpleShop(String name, String address {
    aAddress = address;
    sName = name;
    lstInventory = new ArrayList();
} dTotalAmountSold = "0.0"; B

public String getAddress(){
} Return sAddress; A- should be return

public int getPhoneNumber(){ C- wrong return type
    return sPhoneNumber;
}
B - should be void
public int setPhoneNumber(String phoneNumber){
    sPhoneNumber = phoneNumber;
}

public void addItem(Item item){
    lstInventory.add(item);
}

public int numberOfItems(){ B - should have return statement
    lstInventory.size();
}
}
public boolean sell Item(Item item){ A
    boolean bSold = false; C
    if(lstInventory.contains(items){
        lstInventory.remove(item);
        dTotalAmountSold + item.getPrice();
        bSold = true;
    }
    return bSold; A - should be +=
}
}
}
```

Question 6

Here are some lines of code that in the right order would make up a method to count the occurrences of a letter in a word.

```
if(sWord.charAt(i) == c)
for(int i = 0; i < sWord.length;
i++) return count;
int count = 0;
public int countLetter(String sWord, char
c) count++;
```

Each box represents a placeholder for the lines of code above. Each line of code must be placed in only one of the boxes.

```
{


{

{

}
}

}
```

Question 7A

What are the values of a, b and c after this code is executed?

```
public void int q7A(){
    int a = 3;
    int b = 6;
    int c;
    a += 2;
    b -= 4;
    c = b + a;
}
```

Question 7B

What will this method return for each pairs of inputs shown?

```
public void int q7B(int num1, int num2){
    return num1 % num2;
}
```

num1	num2	returns
17	5	
18	6	

Give a value for each of the two input parameters that would cause the method to return the value

5: num1..... num2.....

Questions 7C, 7D and 7C all have the same instruction:

Complete the table below to show what this method will return for the various values shown.

Questions 7C

```
public boolean q7C(int iValue){
    boolean bValid = false;
    if(iValue>=FIRST_VAL &&
        iVALUE<SECOND_VAL){ bValid = true;
    }
    return bValid;
}
```

iValue	FIRST VAL	SECOND VAL	returns
17	17	2	
18	17	20	
4	3	4	

Question 7D

```
public boolean q7D(int
    iLimit){ int iIndex = 0;
    int iResult = 0;

    while(iIndex <= iLimit){
        iResult += iIndex;
        iIndex ++;
    }
    return iResult;
}
```

iLimit	returns
-1	
3	
0	

Question 7E

```
public int q7E(int[] numbers){
    int iResult = 0;
    for(int i = 0; idx < numbers.length;
        idx++){ if(numbers[idx] > iResult)
        {
            iResult = numbers[idx];
        }
    }
    return iResult;
}
```

numbers	returns
{1,2,3,4,5}	
{20,-10,6,-2,0}	

Question 10A

```
public double method10A(double[]
    numbers){ double num = 0;
    for(int i = 0; i < numbers.length; id++){
        num += numbers[i];
    }
    return num;
}
```

Question 10C

```
public double method10C(int[] numbers, int
    val){ int x = 0;
    int y = numbers.length-
    1; int z, temp;
    boolean switch = false;

    while (!switch && (x <= y){
        z = (x + y)/2;
        temp = numbers[z];
        if(val == temp){
            switch = true;
        }
        else if(val < temp){
            y = z -1;
        }else{
            x = z + 1;
        }
    }
    return switch;
}
```

A conceptual model for reflecting on expected learning vs. demonstrated student performance

Richard Gluga¹ Judy Kay¹ Raymond Lister² Simon³
 Michael Charleston¹ James Harland⁴ Donna Teague⁵

¹ School of IT, University of Sydney, Sydney NSW Australia

² School of Software, University of Technology Sydney, Sydney NSW Australia

³ School of Design Communication and IT, University of Newcastle, Newcastle NSW Australia

⁴ School of Computer Science and Information Technology, RMIT University, Melbourne VIC Australia

⁵ Faculty of Science and Technology, Queensland University of Technology, Brisbane QLD Australia

Abstract

Educators are faced with many challenging questions in designing an effective curriculum. What prerequisite knowledge do students have before commencing a new subject? At what level of mastery? What is the spread of capabilities between bare-passing students vs. the top-performing group? How does the intended learning specification compare to student performance at the end of a subject? In this paper we present a conceptual model that helps in answering some of these questions. It has the following main capabilities: capturing the learning specification in terms of syllabus topics and outcomes; capturing mastery levels to model progression; capturing the minimal vs. aspirational learning design; capturing confidence and reliability metrics for each of these mappings; and finally, comparing and reflecting on the learning specification against actual student performance. We present a web-based implementation of the model, and validate it by mapping the final exams from four programming subjects against the ACM/IEEE CS2013 topics and outcomes, using Bloom's Taxonomy as the mastery scale. We then import the itemised exam grades from 632 students across the four subjects and compare the demonstrated student performance against the expected learning for each of these. Key contributions of this work are the validated conceptual model for capturing and comparing expected learning vs. demonstrated performance, and a web-based implementation of this model, which is made freely available online as a community resource.

Keywords: curriculum, assessment, course content

1 Introduction

To develop an effective teaching and learning plan for a subject that has prerequisites, a lecturer must be aware of the capabilities of the students at the beginning of that subject. That is, the lecturer must have a solid idea of the knowledge and concepts that students have learnt in the previous semester, and the level of mastery achieved. The teaching schedule, lecture topics and learning outcome statements from the previous subject may provide some indication as to the content that was covered, but this does not detail what was actually assessed, how it was assessed and how it was graded. The marking criteria

for the subject might have awarded most marks for rote-memorisation of algorithms and code recipes. On the other hand, perhaps the assessments tested higher-level problem-solving skills using the learnt concepts in unfamiliar scenarios. It is not easy to discern how much of the overall assessment weight was associated to the former as opposed to the latter. The lecturer, however, must be aware of these details in order to develop an effective teaching program based on the capabilities of beginning students.

Likewise, a lecturer must be able to answer the same questions about the teaching and assessments of his or her own subject. That is, which topics and concepts are expected as subject outcomes, and at what levels of mastery are students expected to achieve them? Additionally, what does the assessment design infer or guarantee about the minimal capabilities expected of bare-passing students at the end of the subject, and how does this compare to the aspirational outcomes expected of top-performing students?

Further still, expected outcomes must be validated against actual learning, as demonstrated by student performance, to ensure that the teaching and learning design is realistic. That is, are bare-passing students meeting the minimal expectations? Are top-performing students achieving the aspirational outcomes? If expectations do not align with demonstrated performance, the lecturer must consider why, and what remedial teaching or assessment changes are appropriate for future offerings of the subject.

Taking a whole program perspective, each individual subject is only one in a long sequence of 24 or more in a typical three- or four-year degree. From semester to semester, students must progressively learn new concepts and build upon the concepts previously learnt. So in order to develop an effective program sequence, each subject lecturer must be able to answer these questions about his or her own subject, and about previous subjects in the sequence. The many subject lecturers involved in the teaching of a degree program must thus have a shared and comparable understanding of the outcomes and mastery levels developed throughout the program.

This paper presents a conceptual model for a systematic curriculum mapping and learner modelling approach that enables subject lecturers to design and document the learning goals in a subject, and to compare expected learning with actual performance as demonstrated by student assessment grades. This is done in terms of a syllabus specification that can be used to communicate learning goals across a whole computer science degree program. The conceptual model also formally captures the level of mastery for each topic or outcome assessed, and the academic's

Copyright ©2013, Australian Computer Society, Inc. This paper appeared at the 15th Australasian Computer Education Conference (ACE 2013), Adelaide, South Australia, January-February 2013. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 136, Angela Carbone and Jacqueline Whalley, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

confidence and judgement of reliability for each of these. This model enables academics to systematically answer some of the difficult questions posed above.

2 Background

Sadler (2009) claims that “academic achievement standards is now the key issue. It is what worries a lot of people”. He asks “do the grades that are on students’ transcripts actually mean what they say?” That is, what do assessment marks actually tell us, if we cannot reliably identify what exactly is being assessed, what cognitive skills are required to pass the assessments, and what a bare-passing grade means compared to the highest passing grade. Sadler suggests that “what we need to do is find ways of capturing the standards we want to use, so we can compare students’ work with those standards”. Doing so means that “each grade represents a particular level of competence, knowledge or skill”, and as Sadler put it, “that is the crux of the matter”.

Similar concerns have also been expressed in the computing education (CSEd) research community. Commenting on the Grand Challenges facing computing education, McGettrick (2005) makes several points about important issues relating to the computer science curriculum, including that there is an increasing need for curriculum standardisation and for comparable outcomes. This is due to the continuing globalisation of the workforce, which requires students, educators, employees and employers to have a common vocabulary for describing discipline skills and competence levels. McGettrick observes that “there are different levels of learning as exhibited by the existence of Bloom’s taxonomy of educational objectives (Bloom et al. 1956). These different levels, as well as the associated degrees of commitment required to achieve these levels, need recognition and their consequences understood”. Two of the grand challenges which relate directly to this are:

- Identify very clearly the technical skills ... that students should acquire throughout their program of study in higher education [2.3.2.i]
- Identify and then employ a phased development of all these skills, ensuring that the skill levels are such that graduates are internationally competitive in terms of their skills... [2.3.2.ii]

2.1 Learning Standards in Computer Science

In order to implement the transparency proposed in the previous section, there needs to be an agreed set of learning goals against which to measure student performance. For computer science disciplines within Australia there are several candidate sets of learning goals that might be useful. These include high-level transferable generic graduate attributes (Barrie et al. 2009), national graduate outcomes such as the upcoming TEQSA TLOs (ALTC 2010), international standards such as the ABET-CAC accreditation guidelines (ABET 2011), and fine-grained Syllabus or Body of Knowledge topic and outcome recommendations such as those from the ACS (Gregor et al. 2008) or the ACM and IEEE (ACM/IEEE 2008, 2013).

In this paper we choose to focus on detailed fine-grained syllabus outcomes, and specifically those from the CS2013 Strawman (ACM/IEEE 2013), which lists over 1366 topics and 1041 learning objectives, categorised into 18 top-level Knowledge Areas and 155 Knowledge Units. Out of the 1366 topics, 257 are classified as Tier-1 Core (absolute essentials), 328 as Tier-2 Core (80% minimum coverage expected) and 781 as electives. Whilst Australian computer science degree programs are not formally accredited against this curriculum, most institutions endeavour to be mindful

of and align with these recommendations. Additionally, the ACM/IEEE CS guideline is one of the most comprehensive and widespread Body of Knowledge descriptions of a computer science degree. As such, it provides a common vocabulary for describing and sharing the design of teaching, learning and assessment activities, both among the different subject lecturers within an institution and across institutions within the wider computer science discipline.

2.2 Mastery and Progression in Computer Science

As well as indicating the need for agreed learning standards, both Sadler and McGettrick proposed that students’ level of competence or mastery would need to be measured against such learning standards. Much research has been published about the importance of this in the CSEd community. Lister & Leaney (2003a,b) proposed a criterion-based grading scheme based on Bloom’s Taxonomy (Bloom et al. 1956), where bare-passing students are expected to show competence at the novice levels (Knowledge and Comprehension) while top-performing students should be challenged at the higher levels (Synthesis and Evaluation). Similar uses of Bloom’s Taxonomy to classify the cognitive complexity of programming exercises have been discussed by many others (Reynolds & Fox 1996, Buck & Stucki 2001, Oliver et al. 2004, Burgess 2005, Whalley et al. 2006, Starr et al. 2008, Thompson et al. 2008, Gluga, Kay, Lister, Kleitman & Lever 2012, Simon et al. 2012). Bloom’s Taxonomy is also the recommended medium for specifying mastery in the CS2008 curriculum (ACM/IEEE 2008) and in the ACS ICT Profession Body of Knowledge (Gregor et al. 2008). The new ACM/IEEE CS2013 Strawman has made a slight departure from Bloom’s Taxonomy, proposing instead a new three-level mastery scale, the merits of which are currently under review (Lister 2012).

2.3 Curriculum Mapping

Having found suitable learning standards (the CS2013 Strawman) and a suitable cognitive classification theory (Bloom’s Taxonomy) on which to model our computer science degree programs, we then turned to literature on curriculum mapping as a framework on which to construct our model. English (1988) proposed that an effective approach to curriculum management “should include a planned relationship between the written, taught and tested curricula”. English stated that effective program planning and auditing “should ensure that the written curriculum has planned relationships to the taught curriculum, and that the taught curriculum and written curriculum are related to the tested curriculum”.

English (1978) also stated that “curriculum guidelines, behavioral objectives, course outlines are all descriptions of a future desired condition” and thus “do not represent the actual curriculum applied by individual teachers”. He saw this as a serious problem, labeling curriculum guides and course outlines as the *fictional* curriculum. He stated that “to exercise quality control over curriculum requires the instructional leader or supervisor to know what the *real* curriculum is in his or her subject area” and unless the real curriculum is “known and quantified, it is not possible to understand ... existing gaps or holes” in the program of study. English proposed that “a fairly accurate picture of the real curriculum” must be obtained in order to allow for effective quality control.

Curriculum mapping has been used extensively in K-12 education in the United States (Jacobs 1989, 1991, 1997, 2010). In tertiary education, however, it has been adopted mostly by the medical disciplines

(Willett 2008, Britton et al. 2008, Harden 2001), and more recently to some extent by engineering (Gluga et al. 2010, Wigal 2005) and other professionally accredited disciplines. Examples of such systems in computer science education are limited. One example is the COMPASS system, developed as a Moodle plugin at the University of West Georgia (Abunawass et al. 2004); COMPASS provided mechanisms to link the assessment in each subject to CC2001 topics and learning objectives, at appropriate Bloom mastery levels. This system aimed to answer some of the same questions we identified earlier.

However, COMPASS had a number of limitations. Data entry was “a bit daunting” (Abunawass et al. 2004), in that users had to open external websites to read through syllabus specifications and manually copy over the appropriate topics/outcomes for each assessment mapping. Additionally, “most administrative and review functions require direct interaction with the underlying database using SQL commands”, which meant that visualising the mapped relationships required significant technical expertise and manual data processing. Further still, the system did not integrate with student marks (this was listed as future work, but no further related publications could be found), so there was no way to compare the actual student performance with the intended curriculum design.

3 Conceptual Model

To enable subject lecturers to plan more effective and integrative teaching and learning activities, we have developed a conceptual model for documenting and describing degree programs in terms of well defined learning goals and mastery levels. The conceptual model supports the capture of teaching and learning intention at multiple curriculum stages, based on the ideas introduced by English and others. This model is represented in Figure 1. We define five curriculum stages as follows (leftmost column in the Figure).

- *Recommended Curriculum* – the collection of graduate attributes, national/international learning standards, accreditation competencies and syllabus or body of knowledge recommendations that are relevant for each degree program. A degree program may not need to consider all recommendations, but may aspire to do so for accreditation purposes and recognition purposes. In this paper we focus on fine-grained discipline specific topics and outcomes from an authoritative syllabus, namely the ACM/IEEE Computer Science Curriculum Guidelines (CS2013) (ACM/IEEE 2013).
- *Planned Curriculum* – the structure of a typical three- to five-year degree program, comprising two semesters per year and four core (C) or elective (E) subjects per semester. Each core and elective subject must contribute towards the learning goals from the Recommended Curriculum that the degree program aspires to align with, such as the 1366 topics and 1041 outcomes of the CS2013. Significant planning is required to decide which topics are to be covered in which subjects, and at which levels of mastery, to ensure an effective progressive sequence of study.
- *Practised Curriculum* – the outcomes and learning activities in every subject. The outcomes for the subject are the lecturer’s interpretation of the aims of the subject, based on the learning goals prescribed as part of the program-level Planned Curriculum. These outcomes thus drive the prerequisite knowledge of the subject, and also the design of learning activities such as lecture topics, lab exercises, text readings, etc.

- *Assessed Curriculum* – the learning goals that are actually assessed as part of each individual subject. The Assessed Curriculum is defined by the subject lecturer when creating the assessment exercises for the class. Each assessment question or task may relate to one or more recommended, planned and practised learning goals.
- *Demonstrated Curriculum* – a description of what students have actually learnt as part of a subject or collection of subjects, based on the fine-grained marks associated with each assessment exercise. The Demonstrated Curriculum is a profile of learners in terms of learning goals and mastery levels achieved, based on the marks from each assessed component.

In this paper we focus on the effectiveness of this conceptual model in enabling subject lecturers to describe the Assessed Curriculum and the Demonstrated Curriculum in terms of fine-grained syllabus/body-of-knowledge learning goals and mastery levels. The model supports description and comparison of the expected performance of the bare-passing student vs. the top-performing student vs. demonstrated student performance in terms of these learning goals and mastery levels. The model additionally supports a mechanism for capturing the academic’s confidence as to the reliability of each classification, such that a confidence value may be used to express overall certainty or uncertainty in each of the presented visualisations. These aspects are discussed in greater detail in the following subsections.

3.1 Modelling the Assessed Curriculum

The Assessed Curriculum represents the subject lecturer’s expectations as to what bare-passing students and top-performing students will have learnt, and will be able to demonstrate, at the end of the subject. This is represented in the Assessed Curriculum section of Figure 1 as a collection of exams or assessments designed to measure student learning. Each exam or assessment is broken down into a set of questions or sub-tasks, which are graded separately and may assess different learning goals, at different levels of mastery.

Subjects, exams and questions each have a weight component as a function of performance in the overall degree program. That is, a subject usually has a credit-point value, an exam or assessment has an overall subject weight, and a question or task is worth a certain number of marks. These are important for capturing and calculating the strength of evidence for each modelled learning goal and mastery level, as will be discussed later.

On the right side of the Assessed Curriculum box in Figure 1, we show how learning goals, mastery levels, and other elements are mapped to each assessment question. We label these mappings the *Academic Classifications*. The first of these is a *reliability* score that is associated with each assessment. This score represents the academic’s judgement of how reliable the grades from the classified exam or assessment are considered to be. For example, an academic may feel that an end-of-semester closed-book written final exam, completed under strict supervision, is a fairly accurate representation of a student’s capabilities. On the other hand, a take-home assessment may be considered less reliable as an indicator, as the student is easily able to seek external help in completing it, and thus the final mark may not be as reliable an indicator of the student’s actual capabilities.

The remaining four fields in the Academic Classification box map to each assessment question. The first is a *bare-pass friendly* yes/no flag, which indicates if

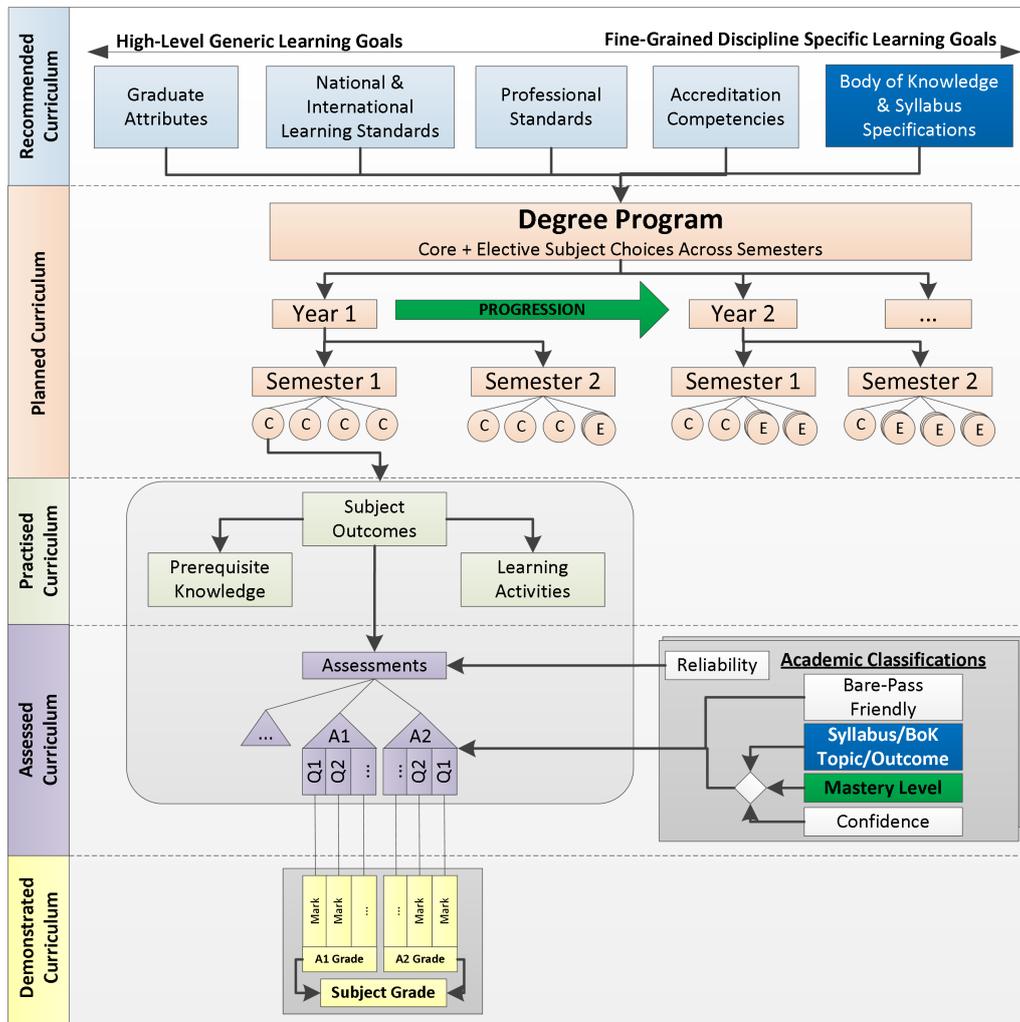


Figure 1: Conceptual model for degree program curriculum design

the academic expects most of the students who finish the subject with a bare-pass mark to be able to earn, say, 70% or more of the marks in that question.

The remaining three mappings (*topic/outcome*, *mastery level*, *confidence*), are stored together as a single sub-classification, and are defined as follows:

1. *Topic/outcome* – a mapping to a relevant topic, outcome or other learning goal from the Recommended Curriculum which is specifically assessed by the question being classified.
2. *Mastery level* – a classification of the cognitive difficulty at which the mapped topic or outcome is assessed (that is, a student would be expected to be operating at this minimal level to answer the question correctly)
3. *Confidence* – a score from 0 to 100 representing the academic's confidence in the validity of this classification.

Multiple instances of such sub-classifications can be made for each question: a question may assess multiple learning goals, each at different levels of mastery (for example, a question may require an advanced understanding of the topic *loops and iteration* but only basic familiarity with *arrays*). The confidence meta-tag can be used to represent any uncertainty in each sub-classification. In some instances it may not be easy to define the mastery level at which a specific topic or objective is being assessed, in which case a low confidence rating can be specified. In other circumstances the academic may feel that even though a topic is being assessed, it is only a small part of the overall question, so again a low confidence rating may

be used to record this as a low evidence mapping.

3.2 Modelling the Demonstrated Curriculum

The Demonstrated Curriculum represents the learning goals and mastery demonstrated by students at the completion of a subject, a set of subjects, or a whole degree program. This is achieved by collecting itemised student marks for each question or task, and using these to compute the achieved level of mastery across the subject/s or program as a whole, or for specific topics or outcomes. These performance scores may then be compared against the Assessed Curriculum design to see how closely they match the modelled expectations.

3.3 Algorithm for Aggregating Classifications

The algorithm for aggregating the assessment question classification data into meaningful forms is as follows.

(i) Calculate the weight of each question (Q_w) as a proportion of the overall subject weight (S) and of the overall degree program (P). Let the credit-point value of the subject be S_{cp} , the weight of each assessment be A_w , and the marks for each question be Q_m , giving $Q_w = (S_{cp}/P_{cp}) \cdot A_w \cdot (Q_m/A_m)$.

(ii) Next, inspect the topic/outcome mappings for each question (TOM). The evidence score for each topic/outcome mapping (TOM_e) is given by the weight of the question (Q_w) divided by the number of topic/outcome mappings for that question (Q_{nTOM}), multiplied by its confidence rating (TOM_{conf}) and the exam/assessment reliability rating (A_{rel}) to give

the final $TOM_e = Q_w \cdot TOM_{conf} \cdot A_{rel} / Q_{nTOM}$.

This now gives a list of evidence scores for each assessed topic/outcome mapping. The sum of all evidence scores for a specific topic or outcome can be used to represent the overall assessment weight associated with that topic across the whole subject, or across the whole degree program. Additionally, each topic/outcome mapping also has a mastery level classification, so we can also sum up all topic/outcome mappings at a specific mastery level to represent the overall assessment weight associated with that level. A third possibility is to sum up all the TOM_e scores for a specific topic/outcome at a specific mastery level.

Further still, we can separate the TOM_e scores into two categories: those from questions that were marked as bare-pass friendly and those from questions intended to distinguish top-performing students. This allows us to create models of the assessed curriculum showing the expected performance of bare-passing students (the minimal standard) vs. the expected performance of top-performing students (the aspirational standard) in relation to the mapped syllabus and mastery levels.

To compute and generate the demonstrated curriculum models, we simply factor the average mark across a set of students for a specific question and multiply this by the topic/outcome evidence score from above. This enables a side-by-side comparison of the expected outcomes of bare-pass students vs. expected outcomes of top-performing students, and vs. actual outcomes of any group of students.

4 User View

We have implemented the conceptual model described above as part of our ProGoSs research system, which aims to enable educators to document the learning across a whole computer science degree and represent it in terms of authoritative curriculum specification. The research presented here is one aspect of the broader BABELnot project (Lister et al. 2012), which aims to document and benchmark the academic standards associated with the core sequence of programming subjects in computer science degrees.

The ProGoSs system allows users to specify the core and elective subjects of a degree program, and then, for each subject, a list of assessments or exams and a sub-list of questions or tasks.

Figure 2 shows the interface for classifying Question 8 from a fictitious final exam in a first semester programming fundamentals subject. The system allows the user to write or copy-paste the actual question text, or to upload an image, or to leave the text empty and instead reference a PDF version of the exam. The question shown in the figure is worth 5 marks, and requires students to *Write a function to return the minimum integer in an array*. Below the question text is the classification meta-data, including the *bare-pass friendly* flag as discussed earlier, and two additional meta-fields for familiarity and estimated time required for students to answer the question. These two fields are not currently used in any further processing.

The lower half of Figure 2 shows the topic/outcome, mastery level, and confidence classifications. In this case, the question was mapped to three topics from the ACM/IEEE CS2013. The large slider on the right is used to quickly set the mastery level for each topic, in this case using Bloom's Taxonomy. All three topics have been mapped at the Application level. Moving the slider left or right moves through the six Bloom levels, with Knowledge to the far left and Evaluation to the far right.

Beneath the mastery sliders is a smaller slider, which can be used to record the user's confidence in

each mapping decision. In this case, all three confidence sliders are set to 100%, indicating the user is very confident in the mappings made. This process is repeated to map all of the questions in a particular exam or assessment.

Additional topic or outcome mappings can be added through a floating dialog editor which allows the user to begin typing a keyword, such as 'parameters', whereupon any matching topics or outcomes from the linked syllabus document will be instantly displayed on the screen. From here, the user can use the sliders to immediately assign a mastery level and confidence, and continue searching for other keywords. The dialog additionally supports manual browsing through the syllabus hierarchy of knowledge areas and knowledge units to select relevant topics. A user may also define his or her own set of topics or outcomes, which may be used in combination with, or instead of, an authoritative curriculum.

The tabs across the top of Figure 2 allow the user to access a range of other functionalities, namely:

- *Overview* – brief description of subject details, typically similar to what appears in a printed handbook.
- *Prerequisites* – mapping of syllabus topics/outcomes and mastery levels that represents expected student knowledge prior to commencing the subject.
- *Assessments* – list of subject exams and assessments, including facility to drill down to individual questions as seen in Figure 2.
- *Dependency checks* – compares the specified prerequisite topics/outcomes to previous subjects in the degree program sequence, allowing the subject lecturer to quickly identify where, and to what extent, each prerequisite topic/outcome was taught and assessed. Similarly, this screen also shows subsequent subjects in the degree program sequence which have prerequisite topics/outcomes that are taught and assessed in the current subject.
- *Program progression* – provides whole-of-program visualisations showing the percentage of planned topic/outcome coverage and planned mastery levels. These are represented via a collection of charts which allow drill-down from high-level knowledge areas to specific topics and outcomes, and to the subjects, assessments and questions where they were assessed. The design and effectiveness of these reports has been presented elsewhere (Gluga, Kay & Lister 2012).

The final tab on the top of Figure 2, *Student Grades*, allows the subject lecturer to view the learning design in terms of the topics/outcomes mapped to exam/assessment questions. It additionally allows the lecturer to import a CSV file containing itemised student grades for each assessment task. Once the grades are imported, the lecturer can generate charts such as the one in Figure 3. These are discussed in the following section as part of our evaluation.

5 Evaluation

To evaluate the conceptual model presented earlier, we initially mapped the final exams from seven core subjects from a computer science degree program offered by an elite Australian ("Go8") university. The questions from each final exam were mapped to the relevant topics/outcomes from the ACM/IEEE CS2013 Strawman draft, at appropriate levels of mastery using Bloom's Taxonomy. This enabled us to generate the Program Progression reports mentioned in the previ-

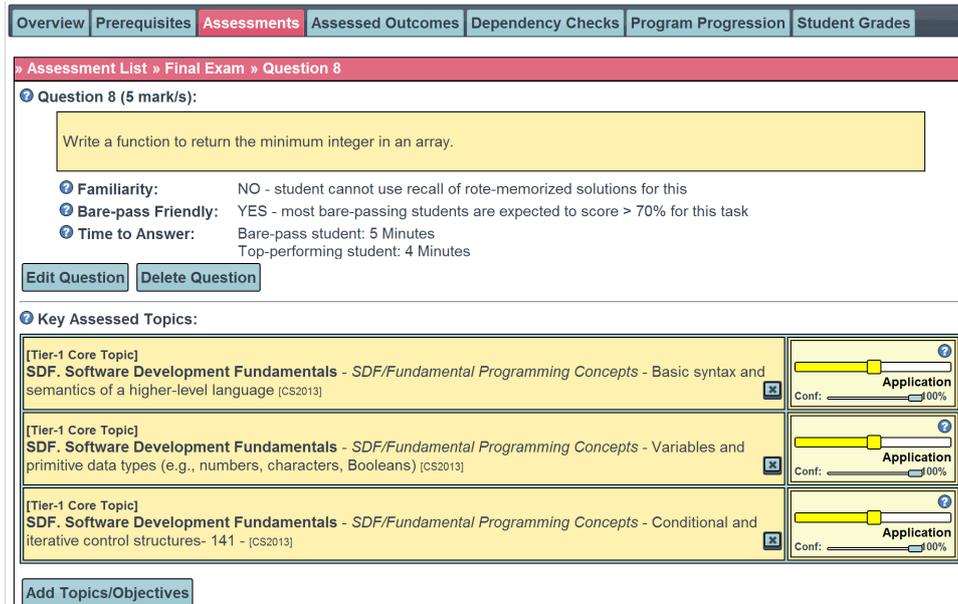


Figure 2: Interface for mapping topics/outcomes and mastery levels to a fictitious question

ous section. This evaluation was described in detail elsewhere (Gluga, Kay & Lister 2012).

The objective of this paper is to evaluate the conceptual model for comparing the assessed curriculum expectations for bare-passing vs. top-performing students against the actual learning achieved, as demonstrated by student grades for each itemised assessment question. To do this, we used the system to code final exams from four programming subjects, each from a different Australian university. The questions from each exam were classified by authors of this paper using the described meta-tags, and validated by an academic involved in the teaching or design of each subject. For one of these four subjects, we also coded the additional three assessments in the subject (two practical tests and one take-home assignment in addition to the final exam). This allowed us to create models of expected learning that took account of the whole of the assessments for the subject.

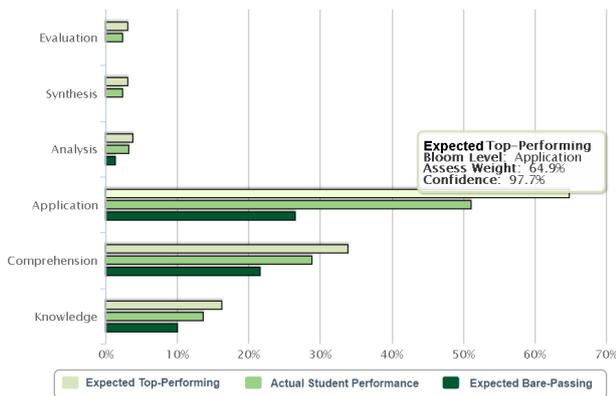


Figure 3: Intended vs. actual performance of top 5% of class in terms of Bloom levels for one subject

For each of these four subjects, we then imported itemised student grades for the four final exams, and also for the three additional assessments for the subject for which we had that information. The numbers of student records imported for the four subjects were 148, 160, 225 and 99. With this data, each lecturer is able to select a subset of their students' grades (for example, the top 10% of students, the bottom 12 students, the 15 students who scored lowest of those who

passed) and the system will generate charts such as the one shown in Figure 3. Along the y-axis are the six Bloom levels, with Knowledge at the bottom and Evaluation at the top.

For each Bloom level, the chart shows three bar values: the top bar is the *expected top-performing* student performance; the middle bar is the *actual student performance* of the selected *subset* of the imported grades; and the bottom bar is the *expected bare-passing* student performance. The x-axis represents the overall subject assessment weight associated with each of the Bloom levels. So, for the example in Figure 3, 65% of the total assessment weight for this subject is mapped at the Application Bloom level for top-performing students (top bar), while the bare-passing students (bottom bar) are expected to achieve 27% of these marks. The selected subset of students (middle bar – in this case the top 5% of the class) achieved 52%. Likewise, the subject had 3% of its assessment weight at the Evaluation level for top-performing students, while bare-passing students were not expected to gain any of those marks. Similarly, the Synthesis and Analysis levels had 3% and 4% of assessment weight for top-performing students and bare-passing students were expected to gain up to 1% of the marks at the Analysis level. It is not our intention to judge whether this mapping of assessment weightings to Bloom levels is appropriate. That is a decision that each university must make for itself. It is merely our intention to make decisions of this sort transparent, so that they can be more readily discussed within an institution.

Hovering the cursor over each bar in the chart brings up a tooltip as seen in Figure 3, which indicates the type of student being modelled (Expected Top-Performing), the Bloom mastery level (Application), the percentage of assessment weight associated with that mastery level (64.9%) and finally the confidence or reliability score for this value (97.7%).

This confidence score is based on the reliability of each assessment and the confidence scores for the topic/outcome mappings as described in the previous section. This provides an indication of the level of accuracy of each of the values. So in the given example, the Application level has an overall reliability score of 97.7%, meaning the classifiers were very confident when mapping questions to the Application level. The Knowledge level, however, had a confidence score of 69%. The final exam in this evaluation was given a re-

liability score of 100%, as it was closed-book and taken under strict supervision. This implies that many of the questions which were mapped at the Knowledge level had low confidence scores associated with them. This is perhaps because the classifier was unsure if the students would use rote learning to answer the question, or reason about the solution using higher cognitive skills. Most of the Application level questions, however, had a 100% confidence rating associated with them.

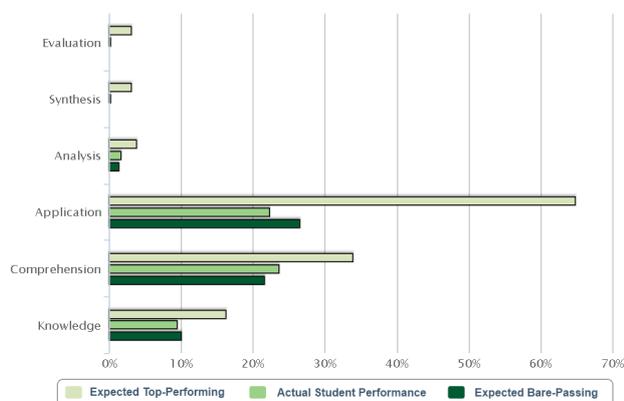


Figure 4: Intended vs. actual performance of bottom 5% of bare-passing students in terms of Bloom levels

The system allows the user to generate this chart for any subset of actual student grades. That is, after importing the student marks, the user may select one or more students to be included in the computation for the middle bar. If only one student is selected, the chart represents the demonstrated curriculum for that individual learner. If, say, five students are selected, the chart shows the demonstrated curriculum as an average across this subgroup. This allows flexible comparison of the expected performance with, say, the actual performance of top students, the actual performance of bare-passing students, the actual performance of the class as a whole, etc.

The middle bar in the chart in Figure 3 shows the actual performance of the top 5% of the class (i.e. the top 12 students, who scored between 82% and 90%). This reveals that the actual top-performing students in this example scored between the expected top and expected bare-passing levels, but closer to the former. Compare this to the chart in Figure 4, which shows the actual performance for the bottom 5% of bare-passing students for the same cohort (that is, the 12 students who scored the lowest marks of 50% or more, which ranged between 50 and 52). This reveals that the actual bare-passing students are scoring marks below the expected bare-passing marks for the Application and Knowledge levels. We could similarly regenerate this chart for the whole class, for a single student, or for any other subset of interest.

The charts in Figures 3 and 4 represent the overall assessment distribution in terms of Bloom levels. A different visualisation allows the user to see the overall assessment distribution in terms of the mapped syllabus topics/outcomes, as shown in Figure 5. The CS2013 topics/outcomes that were mapped to exam questions for this subject are shown along the y-axis. The x-axis shows the overall assessment weight associated with each topic/outcome. The three bars in each series have the same meaning as in the previous two charts, that is, expected top-performing students as the top bar, actual student performance as the middle bar, and expected bare-passing students as the bottom

bar. The image in Figure 5 is cropped to show only the bottom five topic/outcome mappings. The actual chart in the system is scrollable, and in the case of this subject's final exam it shows 43 such mappings. The chart in Figure 5 shows the actual performance of the bottom 12 bare-passing students. For the five topics/outcomes shown, the actual bare-passing performance is very close to the mapped intended bare-passing performance.

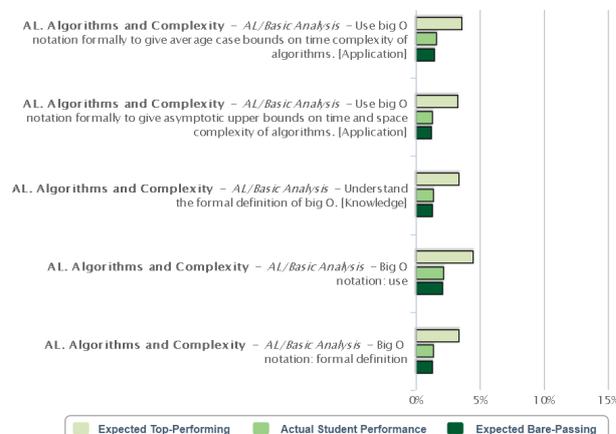


Figure 5: Intended vs. actual performance of bottom 5% of bare-passing students in terms of topics/outcomes (partial)

The charts also support drill-down functionality. Clicking on any of the three bars in the Application series in Figure 3, for example, will bring up a new chart that identifies the assessment weights and reliabilities associated with all of the topics/outcomes that were assessed at the Application level. Further clicking on any bar in the new chart will bring up a dialog showing all of the exam questions that contributed to the clicked-on topic. Likewise clicking on a topic/outcome bar in Figure 5 will bring up a new chart that provides a breakdown of the Bloom levels at which that topic/outcome was assessed, and a further click will bring up the exam questions contributing to those mappings.

5.1 Participant Feedback Results

For each of the four subjects mapped into the system, an academic involved in the teaching or design of that subject was asked first to validate the meta-tags in each question classification, and then to use the charting visualisations described above to compare the assessment design against demonstrated student performance across different groups of students. After doing so, the academics were asked to complete a questionnaire with Likert-scale responses and open-answer feedback commenting on the perceived usefulness of the approach and system implementation. The hypotheses for this evaluation were:

1. Differentiating between the expected outcomes of bare-passing and top-performing students is useful.
2. Comparing expected bare-passing/top-performing outcomes against actual bare-passing/top-performing student outcomes is useful.
3. Visualising the assessment distribution of the subject in terms of mastery levels is useful.
4. Visualising the assessment distribution of the subject in terms of syllabus topics and outcomes is useful.

5. Expressing reliability of classifications is useful.
6. The system interfaces for classifying and visualising information are effective.
7. Academics would consider using the system to model their own subjects and assessments if it were available to them.

The term ‘useful’ in this context is used to capture whether or not participants perceived value in the approach and in the rich reporting interfaces that allowed them to compare expected learning vs. actual learning in their assessments. These were tested using a series of Likert-scale questions that mapped to each of the hypotheses (at least two questions mapped to each hypothesis, with some questions mapping to multiple hypotheses). The scale ranged from 1 (Strongly Disagree) to 5 (Strongly Agree). The average scores for the seven hypotheses were all in agreement (H1=4.25, H2=4.25, H3=4.13, H4=4, H5=3.75, H6=4.63, H7=4.25).

Open-ended feedback by the participating academics was also of high interest. One participant commented in relation to H1 that “I suspect this is something that I always have in the back of my mind when setting assessments...So what this has done is bring these thoughts to the fore and make them explicit on a question-by-question basis”. Increasing the transparency of these assessment design decisions, so that they may be shared across subjects, is an important outcome. One participant commented that “I can see that this would be useful in terms of syllabus design, but once the course is designed and implemented I think the usefulness diminishes”. This may be true to some extent, namely that the approach would be most useful in the initial design of a new subject or new degree program. However, subjects, subject lectures, degree program enrolment rules and even curriculum recommendations do often change. Having the original design decisions explicitly captured will enable more informed restructuring of teaching and learning activities at these points. For example, suppose an academic designs a new subject, including all assessments, so that it aligns with a set of recommended learning goals. What happens when this academic leaves and is no longer responsible for this subject? How does a subsequent lecturer know the implicit reasoning behind the assessment design?

While the participants indicated overall satisfaction with the effectiveness of the system interfaces, some were concerned that the initial data entry may be somewhat time-consuming. Additionally, some participants found that the mapping of questions to the CS2013 topics and outcomes was not always obvious. In particular, a number of exam questions were identified where the primary assessed concepts were not found in the CS2013 specification (e.g. variable scope and static variables). However, overall the participants were satisfied with the use of mastery levels to differentiate between the performances of different student groups. One participant stated “A very useful tool. It has suggested, for instance, a broad difference in the Bloom level that students reach in different bands: below a Credit (65%) for instance, the Application level average performance dips below the performance for Comprehension-level questions. Interesting stuff which might give a good perspective to academics who are hoping to define clearly what it means to be a ‘Credit level’ student vs a ‘Passing’ student.”

6 Discussion

The evaluation presented in this paper used student marks from four subjects, each from a different institution, to validate the conceptual design. For one of the four subjects we were able to import marks for all

of the assessments, not just for the final exam. This provided a more realistic picture of the learning expectations vs. actual performance across that entire subject. When considering only final exams, the generated reports showed very small assessment weight at the higher Bloom levels (Synthesis and Evaluation). This is to be expected, as testing for competence at these higher levels is typically more appropriate in larger design-oriented tasks such as take-home assignments. This was reflected in the subject for which we had student results for all assessments: a large portion of a 20% take-home assignment was classified at the Bloom Synthesis level. The charts for this report thus contrasted with those from the three subjects with only final exams, which had very little emphasis on these higher levels.

However, take-home assignments may have lower reliability scores, depending on the classifier’s judgement. This would thus reduce the confidence scores for these upper Bloom levels, as compared to the stronger reliability for the lower levels assessed in final exams. Appropriate ways in which to interpret these confidence and reliability scores need to be explored. If a topic/outcome has an overall confidence value of less than 50%, what can we claim about the knowledge of the student at the end of the subject? Perhaps the reliability scores are overly pessimistic and need to be raised? Perhaps some of the confidence values associated with each question mapping are too low and need to be revised? Perhaps the subject relies too heavily on less trusted assessment techniques and thus cannot support strong claims about the learning outcomes of the passing student? In any case, the conceptual design and implementation allows the academic to explicitly document and capture these concerns, providing the opportunity to iteratively refine the learning design so as to raise the mastery levels and their confidence values to appropriate levels.

The conceptual design also supports the generation of similar reports and visualisations across a sequence of subjects, or across a full degree program. This may provide very valuable information for degree program quality assurance and accreditation purposes, and for communicating with employers or other stakeholders a more precise picture of graduate capabilities. The main difficulty in doing this is collecting the itemised fine-grained student marks for each individual question in each subject assessment. This may require a change to current assessment processes in some institutions, where typically marks are stored only at a coarse level. For example, only a single mark for a quiz or exam is recorded in the student gradebook system, and after the student completes a subject, these itemised marks are often lost and all that remains is an overall subject mark for each student, which does not provide sufficient information for such analysis.

The validation presented in this paper maps topics and outcomes from the CS2013 Strawman curriculum guideline, which is not formally accredited in Australian computer science degree programs. Perhaps an institution may be more interested in mapping assessment tasks against the ACS Core Body of Knowledge, or the Skills Framework for the Information Age (SFIA 2012) as proposed in the new ACS accreditation guidelines. The skills, attributes and topics listed in these documents are significantly higher-level, so instead of mapping to each individual exam question, it may be more appropriate to classify only at the assessment level, or even the subject level as a whole. Such higher-level attributes and skills are discussed at length in Gluga, Lever & Kay (2012).

The model presented is agnostic of any specific syllabus or body-of-knowledge statement, so it could

instead be used with any internally defined taxonomy of topics or concepts that an institution, department or group of academics decides on as important. Additionally, the model is agnostic of the method by which mastery levels are classified. We have used Bloom's Taxonomy, as it has received significant attention in computing education, but other classification schemes such as neo-Piagetian cognitive development (Lister 2011), the SOLO Taxonomy (Sheard et al. 2008), or any internally defined scheme may work equally well.

The evaluation and discussion thus far have focused on a single offering of each subject. That is, the final exams and student results were from a particular offering of each of the four classified subjects. The charts and reports shown here are thus only a snapshot view of a subject or collection of subjects at a particular point in time. The envisaged use of the system is to model lecturer expectations from a subject offering, then to compare these expectations to actual student performance at the end of the offering, and to take any necessary corrective action in the teaching and learning design or assessment design for the next offering. That is, the conceptual model is intended to be used as a tool for iterative improvement of courses and programs. The snapshot aspect of the data might appear somewhat restrictive, in that each new offering of a course would entail new data. However, it is our experience that while assessment items generally change from one offering to the next, what they assess and how they assess it remain fairly constant; therefore all that is required for a new offering is to check the data for the previous offering and adjust it appropriately.

The primary concern in using the tool for this purpose and in this fashion, as expressed by some of our participants, is the perceived effort required in performing the fine-grained classifications. However, as reported in Gluga, Kay, Lister & Lever (2012), the time taken for mapping a full exam paper is between one to two hours, or slightly more depending on the granularity of questions. Mapping additional assessments from the subject may thus take a further hour or two. An entire subject can thus be reasonably classified by the lecturer of that subject within a single sitting. This would enable very rich long-term models of the curriculum with a modest time investment from each of the 24 or more subject lecturers.

7 Conclusion

To design an effective computer science degree program, subject lecturers need to have a clear understanding of the learning standards that they are to teach and assess, and the capabilities of their students at the beginning and end of each subject. That is, lecturers must know what syllabus topics and outcomes the students have previously learnt, and what mastery level they have attained, in order to design effective teaching, learning and assessment activities that integrate appropriately into the overall degree program sequence. Additionally, lecturers need to be aware of the differences in capabilities between the bare-passing students and top-performing students, to help ensure that neither group is neglected. Likewise, subject lecturers must be able to communicate this knowledge amongst themselves as students progress through the many subjects of a degree. They must additionally be able to support this knowledge with evidence based on actual student grades, such that any unmet expectations can be addressed in future revisions of the curriculum.

To achieve these goals, we have presented a conceptual model that supports the description of subject assessment questions in terms of syllabus topics or outcomes, such as the CS2013, and also in terms

of mastery levels, such as Bloom's Taxonomy. The model additionally supports the importing of student marks to represent the actual Demonstrated Curriculum, which we believe to be important for iterative teaching refinement. A third component of the model is the capture of reliability scores for each assessment task and confidence ratings in each question classification. These are useful for representing the accuracy and reliability of the generated curriculum models, on which important decisions may be based.

We have validated the conceptual model by creating a web-based implementation that enables users to enter all the subject assessment data and to effectively classify each individual question. The system was used to model the Assessed Curriculum based on the final exams of seven core programming subjects from a real computer science degree program. To test the effectiveness of comparing the expected learning outcomes with actual student performance, we imported itemised final exam marks from 632 students across four programming subjects from different institutions. The system was used to aggregate these grades against the question classifications and present a series of charts that allow visualisation of the data from multiple perspectives. Additionally, for one subject we were able to import student marks for all remaining assessments, enabling us to generate realistic reports as to the learning design of that subject as a whole, and to compare that against the Demonstrated Curriculum.

Academics involved in the teaching or delivery of each of the four subjects validated the question classifications and experimented in using the charting visualisations to explore how closely their expectations of bare-passing vs. top-performing students matched the actual student performance at different band levels. Overall, the academics expressed positive interest in using a similar system to document and visualise their subjects and assessments.

The main contributions of this paper are the conceptual model for capturing the learning design and expectations, for comparing these against demonstrated student performance, and for also capturing the reliability of the generated models. The system is freely available to trial online at <http://progoss.com>.

Acknowledgements

This work was supported by a grant from the Australian Government Office for Learning and Teaching.

References

- ABET (2011), 'ABET Computing Accreditation Commission, Criteria for Accrediting Computing Programs', <http://www.abet.org/cac-current-criteria/>. [Last Accessed: 8-11-2012].
- Abunawass, A., Lloyd, W. & Rudolph, E. (2004), COMPASS: a CS program assessment project, *in* 'ACM SIGCSE Bulletin', Vol. 36, ACM, pp. 127-131.
- ACM/IEEE (2008), 'Association for Computing Machinery and the IEEE Computer Society, Computer Science Curriculum 2008 (CS2008)'.
- ACM/IEEE (2013), 'Association for Computing Machinery and the IEEE Computer Society, Computer Science Curricula 2013 (CS2013)'.
- ALTC (2010), 'Engineering and ICT: Learning and Teaching Academic Standards Statement', <http://www.olt.gov.au/resource-engineering-ict-ltas-statement-altc-2010>. [Last Accessed: 08-11-2012].
- Barrie, S., Hughes, C. & Smith, C. (2009), 'The National Graduate Attributes Project: integration and assessment of graduate attributes in curriculum'.

- Bloom, B. S., Engelhart, M. B., Furst, E. J., Hill, W. H. & Krathwohl, D. R. (1956), *Taxonomy of educational objectives. The classification of educational goals. Handbook 1: Cognitive domain*, Longmans Green.
- Britton, M., Letassy, N., Medina, M. & Er, N. (2008), 'A curriculum review and mapping process supported by an electronic database system', *American journal of pharmaceutical education* **72**(5).
- Buck, D. & Stucki, D. J. (2001), JKarelRobot: A case study in supporting levels of cognitive development in the computer science curriculum, in 'Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education', SIGCSE '01, ACM, New York, NY, USA, pp. 16–20.
- Burgess, G. A. (2005), 'Introduction to programming: Blooming in America', *J. Comput. Sci. Coll.* **21**(1), 19–28.
- English, F. (1978), 'Quality control in curriculum development'.
- English, F. (1988), 'Curriculum auditing'.
- Gluga, R., Kay, J. & Lever, T. (2010), Modeling long term learning of generic skills, in V. Aleven, J. Kay & J. Mostow, eds, 'Intelligent Tutoring Systems', Vol. 6094, Springer, pp. 85–94.
- Gluga, R., Kay, J. & Lister, R. (2012), ProGoSs: Mastering the Curriculum, in M. Sharma & A. Yeung, eds, 'Australian Conference on Science and Mathematics Education (ACSME2012)', 18th Annual UniServe Science Conference, UniServe Science, The University of Sydney, NSW 2006, Australia, pp. 92–98.
- Gluga, R., Kay, J., Lister, R., Kleitman, S. & Lever, T. (2012), Coming to terms with Bloom: An online tutorial for teachers of programming fundamentals, in M. de Raadt & A. Carbone, eds, 'Australasian Computing Education Conference (ACE2012)', Vol. 123 of *CRPIT*, ACS, Melbourne, Australia, pp. 147–156.
- Gluga, R., Kay, J., Lister, R. & Lever, T. (2012), A unified model for embedding learning standards into university curricula for effective accreditation and quality assurance, in 'Australasian Association for Engineering Education (AAEE2012)', Vol. [to appear], Melbourne, Australia.
- Gluga, R., Lever, T. & Kay, J. (2012), 'Foundations for modelling university curricula in terms of multiple learning goal sets', *IEEE Transactions on Learning Technologies* p. to appear.
- Gregor, S., von Kinsky, B. & Wilson, D. (2008), 'The ICT profession and the ICT body of knowledge (vers. 5.0)', <http://www.acs.org.au/attachments/ACSCBOKWorkingPaper2008.pdf>.
- Harden, R. (2001), 'Amece guide no. 21: Curriculum mapping: a tool for transparent and authentic teaching and learning', *Medical Teacher* **23**(2), 123–137.
- Jacobs, H. (1989), *Interdisciplinary curriculum: Design and implementation*, ERIC.
- Jacobs, H. (1991), 'Planning for curriculum integration', *Educational Leadership* **49**, n2.
- Jacobs, H. (1997), *Mapping the Big Picture. Integrating Curriculum & Assessment K-12*, ERIC.
- Jacobs, H. (2010), *Curriculum 21: Essential education for a changing world*, Association for Supervision and Curriculum Development.
- Lister, R. (2011), Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer, in J. Hamer & M. de Raadt, eds, 'Australasian Computing Education Conference (ACE 2011)', Vol. 114 of *CRPIT*, ACS, Perth, Australia, pp. 9–18.
- Lister, R. (2012), 'The CC2013 Strawman and Bloom's taxonomy', *ACM Inroads* **3**(2), 12–13.
- Lister, R., Corney, M., Curran, J., D'Souza, D., Fidge, C., Gluga, R., Hamilton, M., Harland, J., Hogan, J., Kay, J. et al. (2012), Toward a shared understanding of competency in programming: an invitation to the babelnot project, in 'Proceedings of the 14th Australasian Computing Education Conference (ACE2012)', Vol. 123, Australian Computer Society.
- Lister, R. & Leaney, J. (2003a), First Year Programming: Let All the Flowers Bloom, in 'ACE '03: Proceedings of the Fifth Australasian Computing Education Conference', Australian Computer Society, Inc., Darlinghurst, Australia, pp. 221–230.
- Lister, R. & Leaney, J. (2003b), 'Introductory programming, criterion-referencing, and Bloom', *SIGCSE Bull.* **35**(1), 143–147.
- McGettrick, A. (2005), 'Grand challenges in computing: Education—a summary', *The Computer Journal* **48**(1), 42–48.
- Oliver, D., Dobebe, T., Greber, M. & Roberts, T. (2004), This course has a Bloom rating of 3.9, in 'Proceedings of the Sixth Australasian Computing Education Conference – Volume 30', ACE '04, Australian Computer Society, Inc., Darlinghurst, Australia, pp. 227–231.
- Reynolds, C. & Fox, C. (1996), Requirements for a computer science curriculum emphasizing information technology: subject area curriculum issues, Vol. 28, ACM, pp. 247–251.
- Sadler, D. (2009), 'Moderation, grading and calibration'.
- SFIA (2012), 'Skills Framework for the Information Age: How SFIA Works', <http://www.sfia.org.uk>. [Last Accessed: 08-11-2012].
- Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E. & Whalley, J. L. (2008), 'Going SOLO to assess novice programmers', *SIGCSE Bull.* **40**, 209–213.
- Simon, Sheard, J., Carbone, A., Chinn, D., Laakso, M.-J., Clear, T., de Raadt, M., D'Souza, D., Lister, R., Philpott, A., Skene, J. & Warburton, G. (2012), Introductory programming: examining the exams, in M. de Raadt & A. Carbone, eds, 'Australasian Computing Education Conference (ACE2012)', Vol. 123 of *CRPIT*, ACS, Melbourne, Australia, pp. 61–70.
- Starr, C. W., Manaris, B. & Stalvey, R. H. (2008), 'Bloom's taxonomy revisited: specifying assessable learning objectives in computer science', *SIGCSE Bull.* **40**(1), 261–265.
- Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M. & Robbins, P. (2008), Bloom's taxonomy for CS assessment, in 'Proceedings of the Tenth Australasian Computing Education Conference – Volume 78', ACE '08, Australian Computer Society, Inc., Darlinghurst, Australia, pp. 155–161.
- Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A. & Prasad, C. (2006), An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies, in 'Proceedings of the 8th Australasian Computing Education Conference – Volume 52', ACE '06, Australian Computer Society, Inc., Darlinghurst, Australia, pp. 243–252.
- Wigal, C. (2005), Managing and aligning assessment knowledge, in 'Frontiers in Education, 2005. FIE'05. Proceedings 35th Annual Conference', IEEE, pp. T3C–13.
- Willett, T. (2008), 'Current status of curriculum mapping in Canada and the UK', *Medical education* **42**(8), 786–793.

A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers

Donna Teague and Malcolm Corney

Queensland University of Technology,
Brisbane, QLD, Australia
Tel: +61 7 3138 2000

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

Alireza Ahadi and Raymond Lister

University of Technology, Sydney,
Sydney, NSW, Australia
Tel: +61 2 9514 1850

Raymond.Lister@uts.edu.au

Abstract

Recent research indicates that some of the difficulties faced by novice programmers are manifested very early in their learning. In this paper, we present data from think aloud studies that demonstrate the nature of those difficulties. In the think alouds, novices were required to complete short programming tasks which involved either hand executing ("tracing") a short piece of code, or writing a single sentence describing the purpose of the code. We interpret our think aloud data within a neo-Piagetian framework, demonstrating that some novices reason at the sensorimotor and preoperational stages, not at the higher concrete operational stage at which most instruction is implicitly targeted.

Keywords: Neo-Piagetian, programming, think aloud.

1 Introduction

Recent theoretical and empirical research indicates that the problems faced by many novice programmers start very early. In empirical work, Corney, Lister and Teague (2011) collected data showing that students who fared poorly on tests held as early as week 3 of semester were less likely to perform well on a code writing task at the end of semester. That empirical result has since been replicated at other institutions (Corney, Teague, Ahadi, and Lister, 2012; Murphy, McCauley, and Fitzgerald, 2012). In theoretical work, Robins (2010) has produced a statistical model to explain the commonly observed bimodal grade distribution, based on the principle that if a student struggles at an earlier point in semester, then the student is more likely to struggle later in semester.

One of the questions used by Corney, Lister and Teague (2011) in their empirical work is shown in Figure 1. They gave this question to students in their fifth week of learning to program in Python. Less than half of their students answered this question correctly. The literature on novice programmers contains three perspectives as to why students might struggle to answer that question, but there are also arguments against each of those perspectives:

- *Programming Misconceptions:* There have been many studies of novice misconceptions (e.g. Du Boulay, 1989). However, Corney, Lister and Teague (2011) used a pre-test to screen out students who had misconceptions about variables, assignment statements and `if` statements.
- *Misunderstanding what was required:* Perhaps the students thought they were required to provide a line-by-line description of the code. Corney, Lister and Teague (2011) argued that this was implausible for three reasons. First, the question was constructed to indicate what type of answer was required. Second, their students had already encountered an "explain in English" question in an earlier test, and had been shown an appropriate sample answer for that earlier question. Third, Corney, Lister and Teague (2011) indicated that most incorrect answers were of the right type, but were simply wrong (e.g. "swap `y1` and `y3`").
- *Poor Self Expression in English:* Perhaps the students knew the correct answer, but could not express that answer? To investigate that possibility, Simon and Snowdon (2011) gave multiple choice versions of code explanation questions to their students, so that students merely had to select the correct answer, rather than express it for themselves. Simon and Snowdon (2011) found that a non-trivial portion of their students selected the wrong option, so poor self expression in English was not the problem for those students.

A more comprehensive refutation of the above three perspectives requires direct observational evidence identifying other reasons why students struggle. In this paper, we provide such evidence, via a think aloud study. It is a small study, involving only seven subjects, but that proved sufficient to identify other reasons why students struggle. Our aim was not quantitative. That is, our aim was not to establish how common those other reasons are in the general population of students.

Observations contrary to the perspectives in the above three bullet points would be even more persuasive if accompanied by an explanatory theory. Lister's neo-Piagetian framework (2011) provides such a theory. In the next section we describe that framework, before presenting our observations from the think aloud study, in terms of that neo-Piagetian framework.

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 $y_1 \geq y_2 \geq y_3$

Figure 1: A question from the week 5 test of Corney, Lister and Teague (2011).

2 The Neo-Piagetian Stages

Classical Piagetian theory focuses on the general intellectual development of children as they mature. Neo-Piagetian theory instead describes the intellectual development through which people progress, regardless of age, as they gain expertise in a specific problem domain, such as chess, or programming.

In the context of neo-Piagetian theory, Lister (2011) described four main stages of cognitive development in the novice programmer, which are (from least mature to most mature) sensorimotor, preoperational, concrete operational and formal operational. The question in Figure 1 is not designed to elicit formal operational

reasoning from novices, so that stage is not discussed any further in this paper. The other three stages are described below, from least mature to most mature.

2.1 The Sensorimotor Stage

Lister (2011) describes a programmer operating at the sensorimotor stage as a novice who is unable to accurately and reliably hand execute (“trace”) small pieces of code to determine the final values in the variables.

There are two broad reasons why some novices do not reliably trace code. One reason is that some novices have misconceptions, which is well documented in the literature (e.g. Du Boulay, 1989).

Another reason is that for some novices the effort of tracing is so great, and the likelihood of a correct trace so small, those novices simply do not wish to trace. In their multinational study, Lister et al. (2004) found that many students who were asked to trace code on a piece of paper returned that paper unmarked. Even among students who do trace, those at the sensorimotor stage find the low level mechanics of conducting a trace so demanding that it does not help them to “see the forest for the trees”. A sensorimotor novice is reluctant to trace a piece of code more than once. If required to generate their own initial data for a trace, the sensorimotor novice tends not to choose data that will help them better understand the code.

Later in this paper, we will present think aloud data for Donald, who demonstrates his understanding of variables, assignment statements and `if` statements, and also demonstrates a willingness to trace, but struggles to organise his tracing in a way that produces reliable results.

2.2 The Preoperational Stage

Preoperational is the next stage after sensorimotor. Novices operating at this stage can trace code accurately and efficiently.

When asked to answer the question in Figure 1, a novice working at the preoperational stage uses an inductive approach. That is, such a novice may perform one or more traces and then make an educated guess based on the input/output behaviour. Later in this paper, we will present think aloud data collected from Lucas and Sierra, who manifest such preoperational reasoning.

Kolikant and Mussai (2008) described how, when given buggy programs to comment upon, some novice programmers viewed the programs as partially correct. This notion of partial correctness is consistent with novices working at the preoperational stage.

2.3 The Concrete Operational Stage

A novice reasoning at the concrete operational stage is capable of deductive reasoning. That is, a novice reasoning at the concrete operational stage should answer the question in Figure 1 quickly and correctly, simply by reading the code. They should not need to perform an explicit, complete written trace to arrive at the answer.

Instead of reasoning in terms of specific values within variables, as the preoperational novice does, the concrete operational novice reasons about code in terms of

constraints on the possible values of variables. For example, after the body of the second `if` statement in Figure 1, the concrete operational novice thinks of `y3` as holding any value satisfying the condition that it is less than the values in both `y1` and `y2`. Furthermore, by the time the code in Figure 1 has completed execution, the concrete operational novice infers that because `y1` is greater than `y2`, and `y2` is greater than `y3` then `y1` is greater than `y3` — that is, the variable values are in descending order. In neo-Piagetian theory, such reasoning is known as transitive inference.

3 Method

3.1 Think Aloud Sessions

We conducted one-on-one think aloud sessions with volunteer students from the first two introductory programming units at QUT during the second half of 2011. The students were asked to complete a series of simple programming tasks while thinking aloud. Ericsson and Simon (1993) developed protocols for eliciting simple unstructured verbalisations. They emphasised the need to minimise the cognitive effort in producing such verbalisations. The goal is to have the subject simply articulate what is going on in their head, rather than formulate an explanation or description for the benefit of the interviewer. Only when the subject is completely focussed on the programming task can we expect to replicate a silent attempt at the same task with the same or similar sequence of thoughts by the subject. Prior to the data collection reported in this paper, all subjects were given the opportunity to practice thinking aloud and to become at ease with the interviewer and familiar with the technology used to record the sessions.

Each think aloud session was recorded using a Smartpen (2011) which digitally captures whatever is written on special dot paper. The digital "pencast" PDFs produced by the Smartpen can be replayed with synchronised visual and audio output using Adobe Acrobat Reader. The sessions can then be replayed during analysis and shared without the need for special technology.

The students participated in think aloud sessions on a more-or-less weekly basis over a semester, each session lasting for roughly 60 minutes.

3.2 The Subjects

IT students at most universities progress through one or more programming units which we will refer to by level. Level 1 refers to the first introductory programming unit with zero pre-requisites. Level 2 units have Level 1 units as a pre-requisite. It is the performance of students at these first two levels that we will discuss in this paper.

To preserve their anonymity, each of the seven students who took part in think aloud sessions chose an alias. Some details about these students at the time of participating in this study are shown in Table 1. Information in that table includes the current programming level at which each student was studying, their Level 1 result, and the week in semester when the think aloud session occurred.

Alias	Level	Level 1 result	Week
Stapler	2	top 21% of cohort	10
Becki	n/a	top 21% of cohort	11
Donald	2	top 69% of cohort	9
John	1	top 17% of cohort	6
Mel	1	top 43% of cohort	6
Lucas	1	top 43% of cohort	6
Sierra	1	top 43% of cohort	6

Table 1: The seven student subjects

Two of the students, Lucas and Sierra, chose to participate in think aloud sessions as a pair, as they were already a pair in their Level 1 unit programming labs.

3.3 The Tasks

Prior to our students completing the task from Corney, Lister and Teague (2011) shown in Figure 1, they each completed tasks which tested their ability to trace code, and explain in plain English the purpose of given code. According to neo-Piagetian stage theory in the programming domain, the inability to trace reliably (or at least to have great difficulty with tracing) is indicative of a novice operating at the sensorimotor stage. The inability to explain code indicates a stage no higher than preoperational.

As Stapler, Becki and Donald were "post Level 1" students, the tasks they were given were a mix of Python (the Level 1 language) and C# which is the language introduced in Level 2. (Even though Becki had no previous exposure to C#, she was confident that she would be able to interpret C# code adequately enough to answer the questions.)

The tracing task given to the Level 1 students is shown in Figure 2. The other students were given the tracing task shown in Figure 3.

To test their ability to reason about code and provide an explanation in plain English, two tasks were used. Level 1 students were given code that swapped the values in two variables using a third as temporary storage as shown in Figure 4. This problem was first used by Corney, Lister and Teague (2011).

Level 2 students (and Becki, who had completed Level 1 but was not doing the Level 2 subject) had to explain more complicated code than just simple assignments. Their code included nested conditional blocks which found the middle of three values. The task not only tested the ability to extract a meaning from the given code, but also tested a student's ability to reason by transitive inference. This task is shown in Figure 5.

Write the values in the variables after the following code has been executed:

```
x = 7
y = 5
z = 3
x = y
z = x
y = z
```

Figure 2: Level 1 tracing task

Write the values of the specified variables after all of the statements have been executed:

```
a = 7
b = 3
c = 2
d = 4
e = a
a = b
b = e
e = c
c = d
```

Figure 3: Level 2 tracing task

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:

```
c = a
a = b
b = c
```

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:

```
j = i
i = k
k = j
```

Sample answer:
It swaps the values in variables i and k

Figure 4: Level 1 Explain in Plain English task

4 Results

4.1 Performance on the Tracing Tasks

John and Mel had no difficulty with the tracing task in Figure 2. They each accurately verbalised and wrote down the changing value of the variable at each assignment. Similarly, Stapler and Becki were both able to perform an accurate trace of the code in Figure 3 without any difficulty. This provides evidence that these four students were operating *at least* at the preoperational stage.

Although Lucas and Sierra eventually traced the code in Figure 2 accurately, there were some initial indications of a misconception about variable assignment. For example, at one point Lucas read the line "y = z", but articulated "y is assigned to z". This could be interpreted as a clear misunderstanding of the direction of assignment, or alternatively as evidence of cognitive overload causing confusion and an inability to accurately articulate their understanding. After some discussion with his partner Sierra, they agreed on the direction of the assignment, and established a somewhat clearer method of articulating this. The greater cognitive effort required

by this pair to correctly trace the code indicates that for them the concepts of variables and assignment were more fragile than for John, Mel, Stapler and Becki. However, their ability to complete the tracing task provides evidence that they may be operating at the preoperational stage.

There are two initialised integer variables in scope: **first** and **second**. If you were asked to describe the purpose of the code below which uses those variables, a good answer would be "*It outputs the highest value.*"

```
if (first >= second) {
    Console.WriteLine(first);
} else {
    Console.WriteLine(second);
}
```

In one sentence that you should write in the empty box below, describe the purpose of the following code, where **first**, **second** and **third** are integer variables which are all initialised and in scope:

```
int maximum = Math.Max(first, second);
maximum = Math.Max(maximum, third);
int minimum = Math.Min(first, second);
minimum = Math.Min(minimum, third);

if (first == minimum) {
    if (second == maximum) {
        Console.WriteLine(third);
    } else {
        Console.WriteLine(second);
    }
} else if (second == minimum) {
    if (first == maximum) {
        Console.WriteLine(third);
    } else {
        Console.WriteLine(first);
    }
} else {
    if (first == maximum) {
        Console.WriteLine(second);
    } else {
        Console.WriteLine(first);
    }
}
```

Sample answer:
Finds the middle integer value

Figure 5: Level 2 Explain in Plain English task

Donald, on the other hand, experienced great difficulty with his tracing task. He demonstrated very poor tracing skill. He lost his way many times over the considerable period of time it took him to complete the task. Donald was unable to keep track of the changing values in the variables and exhibited increasing frustration during his several attempts to complete the trace.

Table 2 summarises our classification of the seven students, on the basis of their performance on these

tracing tasks. Tracing code is a task performed reliably and accurately by a preoperational student, so John, Mel, Stapler and Becki provided evidence for being *at least* at the preoperational stage. The performance of Lucas and Sierra is not so easy to classify, but after much discussion between them they did eventually successfully complete their tracing task, so perhaps they are at the preoperational stage. Donald, on the other hand, showed little capacity to trace. He is probably at the sensorimotor stage.

4.2 Performance on the Explanation Tasks

For those students who we classified as being *at least* preoperational on the basis of the tracing task, a further test is required to determine if they are actually at the concrete stage, which is the next neo-Piagetian stage. Our further testing entailed observing their ability to extract meaning from code — to “explain in plain English”.

John completed the explaining code task shown in Figure 4. He started by writing down the three lines of code that are given in the question and then, after thinking for a short period of time, determined that the code swapped the values in *i* and *k*. We classified his performance as concrete operational.

Mel simply read the question and established that this code was “also swapping”. As Mel did not elaborate on which variables’ values were being swapped, we cannot be absolutely confident that she had established that *i* and *k* were swapped. She could have meant that the code changed the values of all the variables, although the use of the word “swapping” might indicate a more specific function, albeit ambiguous. We made a determination of Mel’s ability to reason about code as being concrete operational, based upon her performance on the problem discussed in Section 4.3 and also on other problems Mel has done, which are not described in this paper.

Stapler carefully read the question in Figure 5 and wrote down the purpose of `maximum` and `minimum` variables. After reading only a small portion of the remaining code, he drew a quick and accurate conclusion about its purpose. Becki also completed this task quickly.

In solving this problem, our four *at least* preoperational stage students all provided evidence suggesting they are operating at the concrete operational stage. They could see the relationships between different elements of a piece of code and were able to explain the overall purpose of that code. On the basis of this evidence, and the evidence from the earlier tracing task, we classify our seven students as operating at the neo-Piagetian stages shown in Table 3.

4.3 Performance on the Question in Figure 1

We now discuss the performance of all seven of our students on the question in Figure 1. The four students listed in Table 3 as being concrete operational also manifested concrete operational reasoning on this problem. Three of those four novices provided a completely correct answer. The exception was John, who nominated ascending order rather than the correct descending order. We regard that as a minor oversight by John.

None of the four students at the concrete operational stage completed a written trace. All but one of them wrote nothing except their answer. The exception was Becki, who wrote down “ $4 < 3$ ” before crossing it out a few seconds later, and then she wrote nothing else except her final answer.

Alias	Neo-Piagetian Stage
Stapler	<i>at least</i> preoperational
John	<i>at least</i> preoperational
Becki	<i>at least</i> preoperational
Mel	<i>at least</i> preoperational
Lucas & Sierra	preoperational
Donald	sensorimotor

Table 2: The manifested neo-Piagetian stages of the seven students on their tracing tasks

Alias	Neo-Piagetian Stage
Stapler	concrete operational
John	concrete operational
Becki	concrete operational
Mel	concrete operational
Lucas & Sierra	preoperational
Donald	sensorimotor

Table 3: The manifested neo-Piagetian stages of the seven students after their Explanation Task

	Students			
	Stapler	John	Becki	Mel
Seconds to read preamble	59	54	31	70
Seconds to read code and write answer	78	179	69	115
Total Time	2 mins 17 secs	3 mins 53 secs	2 mins 10 secs	3 mins 5 secs

Table 4: The time taken by the concrete operational students to answer the question in Figure 1.

Table 4 summarises the time taken by these four concrete operational students to answer the question. The row in that table “Seconds to read preamble” shows the amount of time that elapsed from when each student started reading the question until the student reached “do NOT write that code”. The next row in the table shows the remaining time each student spent on the task.

John took longer than the others to answer the question, because he was briefly puzzled by the purpose

of the third `if` statement. While considering that statement, he uttered “*Why would you repeat that line of code?*” In asking that question, he explicitly manifested concrete operational reasoning.

With the exception of John, the data presented here does not prove that these students were reasoning at the concrete operational stage. It is possible that a novice operating at the preoperational stage might perform a single trace of this code in their head, and then make a correct inductive guess. However, based on the earlier think aloud data we presented for these four students, it is likely these students are also reasoning at the concrete operational stage on this problem. Our intent in presenting this data for these students is not to provide conclusive evidence that they reasoned at the concrete operational stage, but instead to provide a point of reference, a contrast, to the three novices for whom we next present think aloud data – Donald, Lucas and Sierra.

5 Pre-Concrete Operational Students

5.1 Donald – Sensorimotor

The think aloud session described here was Donald’s fifth such one-hour session, so he was familiar with the process by this time.

In his first 60 seconds, Donald read the question aloud, from the beginning down to “do NOT write that code” (i.e. just above the code he needed to describe). While reading aloud, he manifested good English reading and language skills.

He next read the code, for about 35 seconds, without writing anything down. As he read, he articulated the following description of the code:

So ... if y1 is less than y2, code to swap the values in y1 and y2 goes here ... so if it's less than ... then the greater number goes into y1, if y2 less than y3 ... goes here ... here. Okay.

(Note that Donald says “so if it’s less than ... then the greater number goes into y1”. In the subsequent effort to complete the trace, that crucial observation is later forgotten by Donald.)

Donald then began the annotations shown in Figure 6. (The annotations “Line 1”, “Line2” and “Line 3” are not Donald’s work, but were added by the authors of this paper.) After beginning by writing “y1” on Line 1, Donald said “*hypothetically let’s say y1 was 1*”. He then wrote that number above “y1”. After writing out the rest of the code on Line 1, he wrote a “2” above “y2”. He then added the arrow above Line 1, while saying “*So y1 would get number 2*”. From writing the initial “y1” to drawing the arrow, there is little hesitation, and only about 15 seconds elapse.

Donald then began producing line 2 in Figure 6. He first wrote “ $y2 < y3$ ” followed by a “2” above “y2”. That “2” is obscured in the figure by a correction made later (see below), as “2” is not the correct value for y2 after the execution of Line 1. Donald then wrote a “3” above “y3” and drew the arc from “y3” to “y2”. He then sporadically muttered variable names for 15 seconds, before saying “*Wait ... what have I done?*” After a 10 second pause he said “*Oh yeah*”, and corrected his earlier error by writing a “1” over the top of the “2”. He then correctly updated

his annotations to show the values of y2 and y3 being swapped, by writing a “3” above and to the left of “y2” and “1” above and toward the right of “y3”. However, he did not cross out the previous values in those two variables, and that may be the source of his subsequent confusion (as will be described in the next paragraph). He said at this point, “*Probably should write that out clearer, but anyway ...*” From his first annotation on line 2 to his last annotation on that line, 57 seconds elapse, which is much longer than his time to complete Line 1.

Line 1
Line 2
Line 3

Figure 6: Donald’s first and unsuccessful trace.

Donald then began his third line of annotations. He first wrote “y1” and immediately added “2” above it, while uttering “*y1 was now holding number 2*”. He then wrote the remainder of the third line, while uttering “*and y2 is holding ...*” after a pause of several seconds he commented “*it’s a poor way to write it out for myself*”. From his first annotation on line 3 to his last annotation on that line, 20 seconds elapse.

Even though Donald wasn’t able to complete his first trace, the effort of attempting that trace did then lead him to articulate a tentative idea as to what the code does, but it is a wrong idea:

“I think what it’s doing is just swapping them all down, so reversing. So, it swaps, ultimately what’s in y1 with y ... 3? ... Oh my God, if I could do this clearer it would be much easier to figure out.”

Donald then began a second and clearer trace, by writing the annotations shown in Figure 7. He first wrote, on each of the three lines, respectively “y1 = 1”, “y2 = 2” and “y3 = 3”. He then performed a conventional and correct trace, which took 67 seconds.

y1 = 1
y2 = 2
y3 = 3

Figure 7: Donald’s second trace of the code, which was his first successful trace.

Despite having just completed the correct trace shown in Figure 7, he then re-articulates the same wrong idea about what the code does:

“So, what we’ve done is reverse the variables ... swap ... So, let’s say that in a nice elegant sentence: It swaps the variables, the order of the

variables ... no, it reverses the order of the variables? Yeah?"

After a few more seconds of broken muttering, he writes his incorrect answer:

"To reverse the values stored in y1, y2 and y3 and then print them to screen."

At this point, almost six minutes have elapsed since Donald first began reading the question, and almost four minutes have elapsed since he began to read and trace the code, which is slower than the times shown in Table 4 for the concrete operational students.

After Donald indicated to the interviewer that he had finished the question, the interviewer asked Donald to trace the code again, using the initial values $y1 = 2$, $y2 = 1$ and $y3 = 3$. After Donald had written down those three initial values, as shown in Figure 8, he proceeded to perform a correct trace, taking about 70 seconds to do so, including a short pause (perhaps a pause of surprise) when the first `if` block did not cause the values in $y1$ and $y2$ to be swapped. On completing the trace, however, Donald initially maintained that the code *"ended up the same ... as what I originally came up with"*. (His tone of voice, however, may suggest this utterance was more a question to the interviewer than a committed statement, or perhaps even an ironic remark. In any event, he certainly does not articulate at this point an alternative description of the code.) After being challenged on the correctness of that assertion by the interviewer, but without the interviewer hinting any further as to what the correct answer might be, Donald exclaimed:

"Oh! It's ordering them ... um ... so, it's more about, it's not to rev ... hang on ... oh [indecipherable]... rather than to reverse, it would be to, place them from highest to lowest."

The image shows three lines of handwritten text on lined paper. The first line is $y1 = 3$, the second is $y2 = 2$, and the third is $y3 = 1$. The numbers 3, 2, and 1 are written in green ink, while the variables and equals signs are in black ink.

Figure 8: Donald's third trace of the code, using initial values provided by the interviewer.

5.2 Interpretation of Donald's Performance

While Donald did successfully trace the code at his second attempt, his first attempt demonstrates that he struggles to organise his tracing in a way that efficiently produces reliable results. Despite successfully tracing the code at his second attempt and thus having read the code closely, his subsequent answer is really a guess. Even if he had performed a third trace spontaneously, it is not clear whether Donald could have specified for himself suitable initial values that would have tested his guess. Because tracing is a difficult process for Donald, he only tests his guess with a third trace when asked to do so by

the interviewer. It also falls to the interviewer to provide initial values that will test Donald's guess.

When Donald does abstract from the code, his abstractions are localised, so that an abstraction he makes at one moment can be inconsistent with a subsequent abstraction. For example, early in the process of reading the code, Donald says *"so if it's [viz. $y1$] less than [$y2$]... then the greater number goes into $y1$ "*, but that abstraction is forgotten by, and inconsistent with, his subsequent answer, *"reverse the values stored in $y1$, $y2$ and $y3$ "*.

5.3 Lucas / Sierra – Preoperational

Lucas and Sierra elected to work together in think aloud sessions because they were already working as a pair in weekly programming laboratory sessions. Their approach to think aloud sessions reflected their pair-programming laboratory sessions – one of them would write and describe what he was doing, while the other monitored and intervened when he felt it was necessary. This was their third think aloud session, so they were well practiced. In this session, and also in earlier sessions, Lucas and Sierra manifested good language skills.

In their first 60 seconds, Sierra read the question aloud from the beginning down to *"do NOT write that code"* (i.e., just above the code they needed to describe). They then read and discussed the code, for about 50 seconds, without writing anything down. The following is a transcript of that discussion:

Sierra: *If $y1$ is less than $y2$...*

Lucas: *... code to swap the values in $y1$ and $y2$ goes here, yeah.*

Sierra: *If $y2$ is less than $y3$, swap the two.*

Lucas: *Yep. If $y1$ is smaller than $y2$, okay, so if that is smaller than that, their values swap, if that is smaller than that, after that, their values swap, and if that is smaller than that, then their values swap; does that mean they just ... changing the order of the ascending and the descending? Like, for instance, I'm just gonna write it out.*

Lucas then wrote down the first line shown in Figure 9. As he did that, Sierra suggested the initial values shown in the boxes on that line. Next, they collaborated on a smooth and successful trace, which took 42 seconds.

The image shows three lines of handwritten text on lined paper. The first line is $y1 = 2$, the second is $y2 = 1$, and the third is $y3 = 3$. The numbers 2, 1, and 3 are written in green ink, while the variables and equals signs are in black ink. Each number is enclosed in a small green box.

Figure 9: The first trace by Lucas and Sierra.

The following discussion then ensued:

Lucas: *Thus, we swap the order ...*

Sierra: *But we don't want to talk about each line of code. Basically, we are just changing the order of code from ...*

Lucas: *We're changing the integer values, swapping it, but what if ... let's do a "what if" scenario ...*

Sierra: *It's changing it from, basically ascending order to descending order.*

Lucas: [While writing the first line shown in Figure 10.] *If these are the correct integers. I'm just going to try it the other way around, so make y1 three, make y2 two and y1 one.*

They then collaborated on completing another smooth and successful trace, as shown in Figure 10.

y_1	y_2	y_3
$\boxed{3}$	$\boxed{2}$	$\boxed{1}$
3	2	1
3	2	1

Figure 10: The second trace by Lucas and Sierra.

As Lucas wrote the second last line of Figure 10, the conversation continued:

Sierra: *I see what they are doing now. It's like what I said before ...*

Lucas: [While writing the final line in Figure 10] *... and if y..., that's still the same, so it just changes it, from ascending, ah, yeah, from largest to smallest ... it will consistently be that.*

Sierra: *That's what we've got to write.*

Lucas then commences writing their answer while saying it aloud:

Lucas: *"The purpose of the code is to ...*

Sierra: *... change the order? From ...*

Lucas: *Well, not necessarily change it, because if it is already in that order, then it will ... not change it, what's another word?*

Sierra: *Is to, umm, rearrange?*

Lucas: *Yeah, I guess ... to arrange, not rearrange, because that implies again that you are moving the integers.*

Sierra: *Alright.*

Lucas then completes writing their final, correct answer:

"... arrange variables from highest to lowest."

At this point, almost 5 minutes have elapsed since Lucas and Sierra began reading the question, and almost 4 minutes have elapsed since they began to read and trace the code, which is longer than the times for the concrete operational students in Table 4.

5.4 Interpretation of Lucas and Sierra's Performance

Lucas and Sierra are clearly more advanced novices than Donald, as they traced the code faster and accurately. However, they are not as advanced as the four students who manifested aspects of concrete operational reasoning, since Lucas and Sierra felt a need to trace the code with specific values – trace it twice, in fact.

It might be argued that Lucas and Sierra traced the code because they were more cautious than the four concrete operational students. Based on our observations across multiple think aloud sessions, Lucas and Sierra are not more cautious than the others, but it is not necessary to rely on such an argument – the transcript demonstrates that their primary way of thinking about code is in terms of specific values within variables. We elaborate upon this point in the remainder of this section.

Before commencing their first trace, Lucas and Sierra have an hypothesis, "*changing the order of the ascending and the descending?*" which is rearticulated as the first trace proceeds to "*It's changing it from ascending order to descending order*". Note, however, that their hypothesis over-specifies what the code does, as the initial values could be in any order. As their first trace proceeds, the hypothesis they articulate describes the code in terms of the specific values they are tracing. It is not Lucas and Sierra's reliance on an explicit, written trace that indicates they are less sophisticated than the four concrete operational students. Rather, it is their focus on reasoning via specific values in the variables. Their approach is inductive, not deductive.

The inductive nature of their thinking is most evident in their decision to confirm their hypothesis by performing a second trace, rather than by simply reading the code. Their choice of initial values for the second trace demonstrates some abstract thinking, as these values will not be altered if their hypothesis is true. Never the less, they felt a need to trace the code with specific values to confirm their hypothesis, rather than confirming the hypothesis by simply reading the code. Furthermore, when selecting the words for their answer, Lucas argued against using the word "change", because "*if it is already in that order, then it will ... not change it*". While his argument is correct, Lucas was focusing upon their second trace, where the initial values of the variables remained unchanged. He was not making a general argument in terms of all possible initial values.

6 Conclusion

The purpose of this small qualitative think aloud study was to see if we observed problems in novices other than poor English reading/writing skills, misunderstanding the nature of the answer required, and misconceptions about how programs work. Three of our seven subjects (Donald, Lucas and Sierra) did manifest other problems. In neo-Piagetian terms, these three subjects manifested reasoning at the sensorimotor and preoperational stages. Our results are the first direct observational data that is described explicitly in neo-Piagetian terms.

Are Donald, Lucas and Sierra unusually poor students? The grades achieved by these students as shown in Table 1 indicates otherwise. Also, the results

from the earlier quantitative study by Corney, Teague and Lister (2011), and the replication by Murphy, McCauley, and Fitzgerald (2012), suggest that Donald, Lucas and Sierra are not unusual. Furthermore, a multinational study (Lister et al., 2004) established that many students have poor tracing skills at the end of their first semester.

Computing educators need to be more aware of students like Donald, Lucas and Sierra, and should explicitly foster the development of tracing and code comprehension skills in those students. At the very least, our study suggests that when a computing educator encounters a student who struggles to write code, the educator should first check whether that student has adequate tracing and code comprehension skills, before assuming the student is weak at problem solving.

According to constructivist theory, people build new knowledge on the foundation formed by their prior knowledge. We believe that the ability to trace code is the foundation on which abstract reasoning about code is built. That is, while the ability to trace code requires little need to form abstractions of the code, we believe that a novice will not begin to construct correct abstractions in their mind in the absence of the foundational skill of tracing code.

Tracing code is an error prone activity, even for experienced programmers, so the essential skill is not the ability to always get traces exactly right. Instead, we believe the necessary foundation is an efficient strategy for tracing that usually provides a correct answer, perhaps with greater than 50% accuracy, as suggested by Philpott, Robbins and Whalley (2007). While Donald did trace the code correctly at his second attempt, he is an example of a novice who lacks an efficient and well organised strategy for tracing. He did not arrive at a correct explanation until the interviewer intervened to provide initial values for a trace that falsified his initial explanation. Without an efficient tracing strategy, Donald is reluctant to perform more than a single trace, and thus he will struggle to build correct abstractions.

An inability to trace code might explain the bimodal grade distribution that many who teach programming claim to observe (Robins, 2010). Students who have not mastered an effective strategy for tracing code may lack the ability to construct in their mind the abstractions necessary for writing code, and thus can only flounder around, attempting to write code by randomly permuting their code, running it, then repeating that process many times. Those students may form the lower of the two modes in the purported bimodal distribution. Students who have mastered an effective strategy for tracing code have the potential to construct in their minds the necessary abstractions for writing code, and those students might form the upper purported mode.

Acknowledgements

We thank our student volunteers, Becki, Donald, John, Lucas, Mel, Sierra and Stapler. Support for this research was provided by the Office for Learning and Teaching, of the Australian Government Department of Industry, Innovation, Science, Research and Tertiary Education. The views expressed in this publication do not necessarily reflect the views of the Office for Learning and Teaching or the Australian Government.

References

- 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. pp. 95–104.
<http://crpit.com/confpapers/CRPITV114Corney.pdf>
- Corney, M., Teague, D., Ahadi, A. and Lister, R. (2012): *Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions*. Fourteenth Australasian Computing Education Conference (ACE 2012), Melbourne, Australia. pp. 77–86.
<http://crpit.com/confpapers/CRPITV123Corney.pdf>
- Du Boulay, B. (1989): Some difficulties of learning to program. In Soloway and Spohrer (eds), *Studying the Novice Programmer*. Lawrence Erlbaum. pp. 283–300.
- Ericsson, K. A., and Simon, H. A. (1993): *Protocol Analysis: Verbal Reports as Data*. Cambridge, MA: Massachusetts Institute of Technology
- Kolikant, Y. and Mussai, M. (2008): *So my program doesn't run! Definitions, origins, and practical expressions of students' (mis)conceptions of correctness*. *Computer Science Education*, **18**(2): 135–151.
- Lister R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. (2004): *A Multi-National Study of Reading and Tracing Skills in Novice Programmers*. *SIGCSE Bulletin* **36**, 4 (June), pp. 119–150.
<http://doi.acm.org/10.1145/1041624.1041673>
- Lister, R. (2011): *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. Thirteenth Australasian Computing Education Conference (ACE 2011), Perth, Australia. pp. 9–18.
<http://crpit.com/confpapers/CRPITV114Lister.pdf>
- LiveScribe (2011): Retrieved October 24, 2012, from <http://www.smartpen.com.au/>
- Murphy, L., McCauley, R. and Fitzgerald, S. (2012): *Explain in Plain English' Questions: Implications for Teaching*. SIGCSE'12, Feb 29–Mar 3, Raleigh, NC, USA.
- Philpott, A, Robbins, P., and Whalley, J. (2007): *Assessing the Steps on the Road to Relational Thinking*. 20th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ'07), Port Nelson, New Zealand. p. 286.
- Robins, A. (2010): Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education*, **20**: 1, pp. 37 – 71.
- Simon and Snowdon, S. (2011): *Explaining Program Code: Giving Students the Answer Helps – But Only Just*. Seventh International Computing Education Research Workshop (ICER 2011), Providence, Rhode Island, pp. 93–99.

What vs. How: Comparing Students' Testing and Coding Skills

Colin Fidge¹Jim Hogan¹Ray Lister²

¹School of Electrical Engineering and Computer Science,
Queensland University of Technology, Brisbane, Qld, Australia

²Faculty of Engineering and Information Technology,
University of Technology Sydney, Sydney, NSW, Australia
{c.fidge, j.hogan}@qut.edu.au, Raymond.Lister@uts.edu.au

Abstract

The well-known difficulties students exhibit when learning to program are often characterised as either difficulties in understanding the problem to be solved or difficulties in devising and coding a computational solution. It would therefore be helpful to understand which of these gives students the greatest trouble. Unit testing is a mainstay of large-scale software development and maintenance. A unit test suite serves not only for acceptance testing, but is also a form of requirements specification, as exemplified by agile programming methodologies in which the tests are developed *before* the corresponding program code. In order to better understand students' conceptual difficulties with programming, we conducted a series of experiments in which students were required to write both unit tests and program code for non-trivial problems. Their code and tests were then assessed separately for correctness and 'coverage', respectively. The results allowed us to directly compare students' abilities to characterise a computational problem, as a unit test suite, and develop a corresponding solution, as executable code. Since understanding a problem is a pre-requisite to solving it, we expected students' unit testing skills to be a strong predictor of their ability to successfully implement the corresponding program. Instead, however, we found that students' testing abilities lag well behind their coding skills.

Keywords: Learning to program; unit testing; object-oriented programming; program specification

1 Introduction

Failure and attrition rates in tertiary programming subjects are notoriously high. Many reasons have been suggested for poor performance on programming assignments, including an inability to fully understand the problem to be solved and an inability to express a solution in the target programming language.

To help determine whether students do poorly on programming tasks due to an incomplete understanding of the problem or an inability to write a solution, we conducted a series of experiments in which these two aspects of programming were evaluated separately. Students in second and third-year programming classes were given assessable assignments which required them to:

1. unambiguously and completely characterise the problem to be solved, expressing the requirements as a unit test suite; and then
2. produce a fully-functional computational solution to the problem, expressed as an object-oriented program.

Both parts of the assignments carried equal weight. By directly comparing how well each student performed on each of these tasks we aimed to see if there was a clear relationship between students' ability to say *what* must be done versus their ability to say *how* to do it.

Since understanding a problem is a necessary prerequisite to solving it, our expectation was that students would inevitably need to do well on problem definition before succeeding in coding a solution. We therefore designed assignments which allowed us to produce independent marks for code functionality and test coverage, enabling students' skills in these two distinct stages of large-scale object-oriented programming to be contrasted.

2 Related and previous work

Our overall goal is to help expose the underlying reasons for students' difficulties in learning how to develop program code. In particular, we aimed to distinguish students' abilities to both describe and solve the same computational problem, using 'test-first' programming as a basis. We did this with classes of students who had already completed two previous programming subjects, so they had advanced beyond the basic problems of developing imperative program code. As a clearly-defined test-first programming paradigm we used 'test-driven development' (Beck 2003), in which unit tests are developed first and the program code is then extended and refactored in order to pass the tests.

The academic role of unit testing in general, and test-first programming in particular, has already received considerable attention. For instance, an early study by Barriocanal et al. (2002) attempted to integrate unit testing into a first-year programming subject. They made unit testing optional to assess its take-up rate and found that only around 10% of students voluntarily adopted the approach (although those who did so reported that they liked it).

In another early study, Edwards (2004) advocated the introduction of testing into student assignments as a way of preventing them from following an ad hoc, trial-and-error coding process. He describes a marking tool called Web-CAT which assesses both program code correctness and unit test coverage. (We do the same thing, but developed our own UNIX scripts for this purpose.) However, where we kept students' code correctness and test coverage marks separate, so that

we could perform a correlation analysis on them, Edwards (2004) combined the marks to produce a single composite score for return to the students.

In a more recent study of unit testing for first-year students, Whalley & Philpott (2011) assessed the advantages of supplying unit tests to students. (We do likewise in our first-year programming subject, giving students unit tests but not expecting them to write their own.) Using a questionnaire they concluded that although it was generally beneficial for students to apply the ‘test-early’ principle, a minority of students still struggle to understand the concept.

In an earlier study, Melnick & Maurer (2005) surveyed students’ perceptions of agile programming practices, and found that students failed to see the benefits of testing and believed that it requires too much work. In our experience, there is no doubt that test-first programming requires considerable discipline from the programmer and can be more time-consuming than ‘test-last’ programming. Nevertheless, we have found that this extra effort is compensated for by a better quality product, as have many other academics (Desai et al. 2008).

Given students’ reluctance to put effort into unit testing, Spacco & Pugh (2006) argued that testing must be taught throughout the curriculum and that students must be encouraged to “test early”. Like us they did this by assessing students’ test coverage and by keeping some unit tests used in marking from the students. (In fact, in our experiments we did not provide the students with any unit tests at all. Instead we gave them an Application Programming Interface to satisfy, so that they had to develop all the unit tests themselves.)

Keefe et al. (2006) aimed to introduce not just test-driven development into first-year programming, but also other ‘extreme’ programming concepts such as pair programming and refactoring. (These principles are also introduced in our overall curriculum, but not all during first year.) Their survey of students found a unanimous dislike of test-driven development, and they concluded that students find it a “difficult” concept to fully appreciate. (In our course we introduce students to unit testing in first year, but don’t get them to write their own test suites until the second and third-year subject described herein. At all levels, however, our experience is also that there is considerable resistance to the concept from novice programmers.)

Like many other researchers, Janzen & Saiedian (2007) found that mature programmers were more likely to see the benefits of test-driven development. Given a choice they found that novice programmers picked test-last programming in general. In a subsequent study they also found that students using test-first programming produced more unit tests, and hence better test coverage, than those using test-last programming, but that students nevertheless still needed to be coerced to follow test-first programming (Janzen & Saiedian 2008). (Both of these findings are entirely consistent with our experiences. Although our own academic staff can clearly see the benefits of test-first programming, our students must be coerced into adopting it.)

Most recently, using attitudinal surveys, Buffardi & Edwards (2012) again found that students did not readily accept test-driven development, despite the fact that those students who followed the approach produced higher quality code. (They report that they taught test-driven development but did not assess it directly. By contrast we directly assessed our students’ unit testing skills.)

Thus, while there have been many relevant studies, none has presented a direct comparison of unit testing and program coding skills as we do below. Fur-

thermore, while we focussed on students with some programming experience, most previous studies have considered first-year students only.

3 Programming versus unit testing

In traditional “waterfall” programming methodologies, program code is developed first, followed by a set of acceptance tests. Modern “agile” software development approaches reverse this sequence (Schuh 2005). Here the tests are written first, in order to define what the program code is required to do, and then corresponding program code is developed which passes the tests. In the most extreme form, *test-driven development* involves iteratively writing individual unit tests immediately followed by extending and refactoring the corresponding program code to pass the new test (Beck 2003). This is usually done in object-oriented programming, where the “unit” of testing is one or more methods. Advantages claimed for this approach to software development include the fact that a “working” version of the system is available at all times (even if all the desired functionality has not yet been implemented), that it minimises administrative overheads, and that it responds rapidly to changing customer requirements.

Most importantly for our purposes, a suite of unit tests can be viewed as a *specification* of what the corresponding program is required to do. It defines, via concrete examples, what output or effect each method in the program must produce in response to specific inputs. For each method in the program, a well-designed unit test suite will include several representative examples of ‘typical’ cases, which validate the method’s ‘normal’ functionality, as well as ‘extreme’ or ‘boundary’ cases, which confirm the method’s robustness in unusual situations (Schach 2005, Astels 2003). Overall, a unit test suite must provide good *coverage* of the various scenarios the individual methods are expected to encounter. (However, unit testing does not consider how separate program modules work together, so it is usually followed by *integration* testing.)

Test-driven software development thus produces two distinct artefacts, a unit test suite and corresponding program code. The first defines *what* needs to be done and the second describes *how* this requirement is achieved computationally. Having taught object-oriented programming and test-driven development for several years, we realised that these two artefacts can be assessed separately, giving independent insights into students’ programming practices.

4 The experiments

To explore the relationship between students’ testing (specification) and coding (programming) ability, we conducted a series of four experiments over two semesters. This was done in the context of a *Software Development* subject for second and third-year IT students. (The classes also contained a handful of postgraduate students, but too few to have a significant bearing on the results.) The students enrolled have typically completed two previous programming subjects, meaning that they have already advanced beyond simple imperative coding skills and are now concerned with large-scale, object-oriented programming.

The *Software Development* subject focuses on tools and techniques for large-scale program development and long-term code maintenance. Topics covered include version control, interfacing to databases, software metrics, Application Programming Interfaces, refactoring, automated builds, etc. In particu-

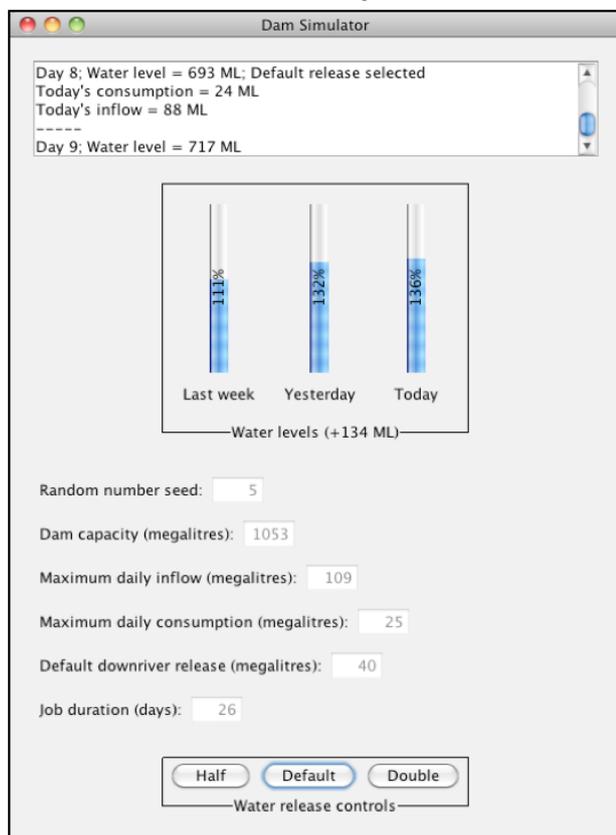


Figure 1: User interface for the solution to Assignment 2a

lar, the role of unit testing is introduced early and is used throughout the semester. Test-driven development is also introduced early as a consistent methodology for creating unit test suites. The illustrative programming language for the subject is Java, although the concepts of interest are not Java-specific.

Each semester's assessment includes two non-trivial programming assignments. Both involve developing a unit test suite and a corresponding object-oriented program, each worth approximately equal marks. The first assignment is individual and the second is larger and conducted in pairs. In the first assignment the students are given a front-end Graphical User Interface and must develop the back-end classes needed to support it. In the second assignment the student pairs are required to develop both the GUI and the back-end code.

For example, one of the individual assignments involved developing classes to complete the implementation of a 'Dam Simulator' which models the actions involved in controlling water levels in a dam. (This topical assignment was introduced shortly after the January 2011 Brisbane floods, which were exacerbated by the overflow of the Wivenhoe dam.) The simulator models the effects on water levels of randomly-generated inflows and user-controlled outflows over a period of time. The user acts as the dam's operator and can choose how much water to release into the downstream spillway each day. The simulation ends if the dam overflows or runs dry.

(This assignment, and the 'Warehouse Simulator' described below, are examples of 'optimal stopping' problems, in which the challenge is to optimise the value of a certain variable, such as a dam's water level, despite one or more influencing factors, such as rainfall, being out of the user's control (Ferguson 2010). We find that these problems make excellent assignment topics because they can form the basis

Method Detail

addEntry

```
void addEntry(Integer newEntry)
    throws SimulationException
```

Adds a new entry to the log. Existing entries are shifted along one place in the window accordingly.

The characteristic feature of the `Log` type is that entries in the series are *indexed in reverse*, using non-positive values, in accordance with the notion that the type implements a log of *past* values. Thus the most recent entry is at index 0, the previous entry is at index -1 , etc. For a log with window size N the oldest entry stored is at index $-N + 1$.

Parameters:

`newEntry` - the new value to be added to the log

Throws:

[SimulationException](#) - if the given value is not in the range 0 to `maxEntry`, inclusive

Figure 2: Example Javadoc specification of a method to be implemented in Assignment 2a

of game-like simulations which are popular with the students.)

The students were given the code for the user interface (Figure 1), minus the back-end calculations. (As per civil engineering convention, the dam's 'normal' water level is half of its capacity, which is why the meters in Figure 1 go to 200%.) To complete the simulator they were required to develop two classes and their corresponding unit tests. One class is used to keep a daily log of water levels in the dam and the other implements the effects of the user's inputs.

The necessary classes and methods to be implemented were defined via Java 'interface' classes, described in standard 'Javadoc' style. For instance, Figure 2 contains an extract from this specification, showing the requirements for one of the Java methods that must be implemented for the assignment. In this case the method adds a new entry to the log of daily water levels. It must check the validity of its given arguments, add the new entry to the (finite) log, and keep track of the number of log entries made to date. Our anticipated solution is the Java method shown in Figure 3.

Each of these Java functions must be accompanied by a corresponding suite of 'JUnit' tests (Link 2003), to ensure that the method has been implemented correctly and to document its required functionality. For instance, Figure 4 shows one such unit test which confirms that this method correctly maintains the number of log entries made to date. The test does this by instantiating a new log object, adding a fixed number of entries into it, and asserting at each step that the number of entries added to the log so far equals the number of entries reported by the log itself. In general there will be several such unit tests for each method implemented—it is typically the case that a comprehensive unit test suite will be much larger than the program code itself. In our own solution we had seven distinct unit tests like the one in Figure 4 to fully define the required properties of the method in Figure 3.

Since the deliverables for these assignments included both unit tests and code, we determined to exploit the opportunity to directly compare how well students performed on unit testing and coding. We considered only those parts of the assignments where both unit tests and code must be produced. (The GUI code in the two pair-programming assignments

```

public void addEntry(Integer newValue) throws SimulationException {
    // Sanity checks
    if (newValue < 0)
        throw new SimulationException("New log entry " + newValue + " too small");
    if (newValue > maxEntry)
        throw new SimulationException("New log entry " + newValue + " too big");
    // Remove oldest entry, if window is full
    if (waterLog.size() == windowSize)
        waterLog.remove(windowSize - 1);
    // Add new entry and keep count
    waterLog.add(0, newValue);
    numberOfEntries += 1;
}

```

Figure 3: Example of a Java method to be implemented in Assignment 2a

```

@Test(timeout = maxWait)
public void NumEntriesIsCorrect() throws SimulationException {
    testLog = new WaterLog(windowSizeSmall, maxEntryLarge);
    // Add entries up to, but not exceeding, the window size
    for (Integer numEntered = 1; numEntered <= windowSizeSmall; numEntered++) {
        // Add an entry
        testLog.addEntry(smallEntryOne);
        // Confirm that number entered equals number reported by the log
        assertEquals(numEntered, testLog.numEntries());
    }
}

```

Figure 4: Example of a JUnit test for the method in Figure 3

Table 1: Statistics for the four assignments

	Assignment			
	1a	1b	2a	2b
Number of classes to implement (excluding GUIs)	2	10	2	6
Number of methods to implement (excluding GUIs)	16	25	12	26
Number of unit tests in our 'ideal' solution	66	63	45	53
Number of assignments submitted and marked	170	83	217	113

was not accompanied by unit tests and was marked manually. Nor did we consider marks awarded for 'code quality' in the experiment.) At the time the experiment was conducted we had four assignments' worth of results to analyse, two from each semester (Table 1). In all four cases the students were required to implement both program code and unit tests and it was emphasised that these two parts were worth equal marks.

- **Assignment 1a:** First semester, individual assignment. This assignment involved completing the back-end code for a 'Warehouse Simulator' which models stock levels in a warehouse. A Graphical User Interface was supplied and students needed to complete a 'ledger' class, to track expenditure, and a 'transactions' class, to implement user-controlled stock buying and randomly-generated customer supply actions.
- **Assignment 1b:** First semester, paired assign-

ment. This assignment involved developing the GUI and back-end code for a 'Container Ship Management System' which models loading and unloading of cargo containers on the deck of a ship, subject to certain safety and capacity constraints. Classes were needed for different container types (refrigerated, dry goods, hazardous, etc) and for maintaining the ship's manifest. (There were a large number of classes and methods in this assignment, but most were trivial subclasses just containing a constructor and one or two additional methods.)

- **Assignment 2a:** Second semester, individual assignment. This was the 'Dam Simulator' assignment described above.
- **Assignment 2b:** Second semester, paired assignment. This assignment involved developing the GUI and back-end code for a 'Departing Train Management System' which modelled the assembly and boarding of a long-distance passenger train. Classes were needed to model the shunting of individual items of rolling stock (including an engine, passenger cars, freight cars, etc) to assemble a train and then simulate boarding of passengers.

In each case the students were expected to follow the test-driven development discipline. For each functional requirement they were expected to develop a JUnit test (such as that in Figure 4) and then extend and refactor their Java program code (like that in Figure 3) until all the system requirements were satisfied. During individual Assignments 1a and 2a the lone student was expected to alternate roles as 'tester' and 'coder'. Pair-programming Assignments 1b and 2b allowed each student to adopt one of these roles at a time.

UNIX shell test scripts were developed to automatically mark the submitted assignments. This was

done in two stages, to separately assess the students' program code and unit test suites.

- **Code functionality:** The students' classes were compiled together with our own 'ideal' unit test suite. (As explained below, this often exposed students' failures to match the specified API.) Our unit tests were then executed to determine how well the students had implemented the required functionality in their program code. The proportion of tests passed was used to calculate a 'code functionality' mark and a report was generated automatically for feedback to the students.
- **Test coverage:** For each of our own unit tests we developed a corresponding 'broken' program which exhibited the flaw being tested for. To assess the students' unit test suite against these programs, their tests were first applied to our own 'ideal' solution program to provide a benchmark for the number of tests passed on a correct solution. Then the students' unit tests were applied to each of our broken programs. If fewer tests were passed than the benchmark our marking script interpreted this to mean that the students' unit tests had detected the bug in the program. (This process is not infallible since it can't tell *which* of the students' tests failed. Nevertheless, we have found over several semesters that it gives a good, broad assessment of the quality of the students' unit test suites.) The proportion of bugs found was used to calculate a 'test coverage' mark and a feedback report was generated automatically.

These marking scripts needed to be quite elaborate to cater for the complexity of the assignments and to allow for various problems caused by students failing to follow the assignments' instructions. The marking scripts ultimately consisted of well over 400 lines of (commented) UNIX bash code (excluding our own ideal solution and the broken programs needed to assess the students' tests).

A particularly exasperating problem encountered during the marking process was the failure of a large proportion of students to accurately implement the specified Application Programming Interface and file formats, typically due to misspelling the names of classes and methods, failing to throw required exceptions, adding unexpected public attributes, or using the wrong types for numeric parameters. This was despite the teaching staff repeatedly emphasising the need to precisely match the specified API as an important aspect of professional software development. In particular, during marking of Assignment 2a it was found that fully half of the submissions failed to match the API specification, and therefore could not be compiled and assessed. In order to salvage some marks for these defective assignments, those that could be easily corrected by, for example, changing the class and method signatures, were fixed manually. In the end 97 assignments, which accounted for 45% of all submissions, were corrected manually and re-marked. Penalties were applied proportional to the extent of correction needed. Ultimately, however, this large amount of effort produced little difference since the re-marking penalties sometimes outweighed the additional marks gained!

Another practical issue noted by Spacco & Pugh (2006), and confirmed by our own experiences, is the difficulty of developing unit tests that uniquely identify a program bug. In practice a single program bug is likely to cause multiple tests to fail. We found when setting up our automatic marking environment, including our own unit test suite (for assessing the students' code) and the suite of 'broken' programs (for

Table 2: Measures of central tendency, spread and correlation for code functionality and test coverage

	Assignment			
	1a	1b	2a	2b
Code functionality mean	82	84	79	91
Code functionality median	86	93	86	96
Code functionality standard deviation	20	20	21	12
Test coverage mean	69	79	45	63
Test coverage median	67	92	52	69
Test coverage standard deviation	23	25	29	24
Functionality versus coverage correlation	0.64	0.76	0.70	0.55

assessing the students' unit tests), that it was impossible to achieve a precise one-to-one correspondence between code bugs and unit tests. While frustrating for us, this did not invalidate the marking process, however.

Moreover, one of the risks associated with this kind of study is threats to 'construct validity' (Arisholm & Sjöberg 2004), i.e., the danger that the outcomes are sensitive to different choices of code functionality and test coverage measures. Nevertheless, we believe our analysis is quite robust in this regard because each of the individual tests in our 'ideal' unit test suite was directly developed from a specific functional requirement clearly described in the Javadoc API specifications, and each of the broken programs was introduced to ensure that a particular unit test was exercised. This very close functional relationship between requirements, code features and unit tests left little scope for arbitrary choices of unit tests or broken programs against which to assess the students' assignments.

5 Experimental results

The marks awarded for code functionality and test coverage were normalised to percentages and are summarised in Table 2. (These values exclude marks awarded for 'code presentation' and for the front-end GUIs in Assignments 1b and 2b.) It is obvious that the average marks for test coverage are well below those for code functionality, which immediately casts doubt on our assumption that students' unit testing abilities would be comparable to their programming skills. Furthermore, while a standard correlation measure (last row of Table 2) shows some correlation between the two sets of marks, it is not very strong.

Further insight can be gained by plotting both sets of marks together as in Figures 5 to 8. Here we have sorted the pairs of marks firstly by code functionality, in blue, followed by test coverage, in red. Assignments 1a and 1b used a coarser marking scheme than was used for Assignments 2a and 2b, thereby accounting for their charts' 'chunkier' appearance, but the overall pattern is essentially the same in all four cases. At each level of achievement for code functionality there is a wide range of results for test coverage. This pattern is especially pronounced in Figure 7.

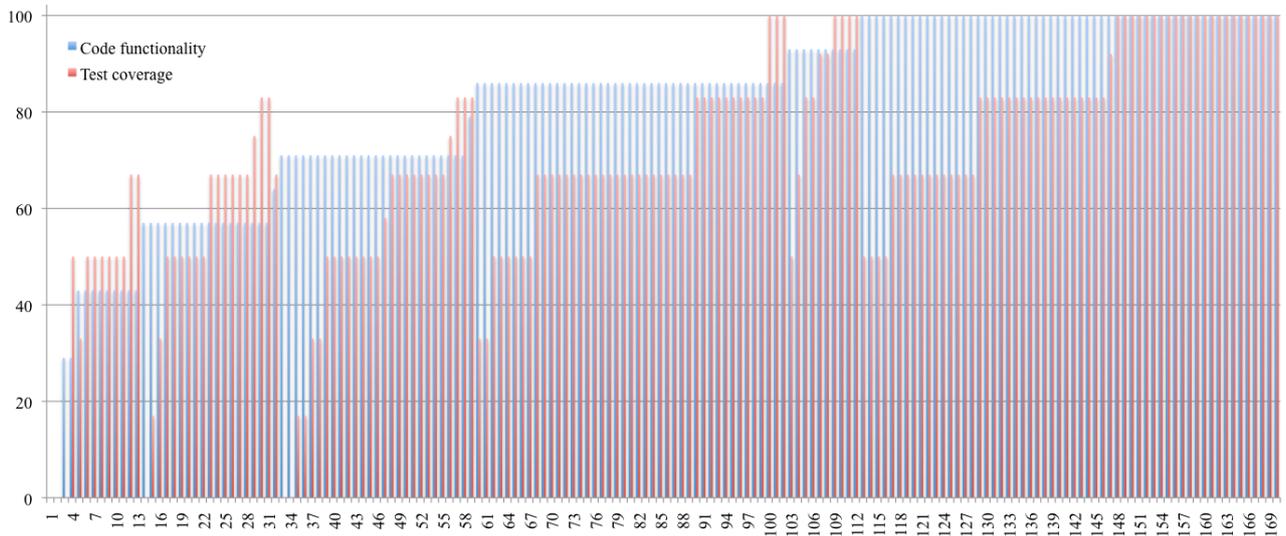


Figure 5: Comparison of marks for code functionality and test coverage, Assignment 1a, 170 submissions

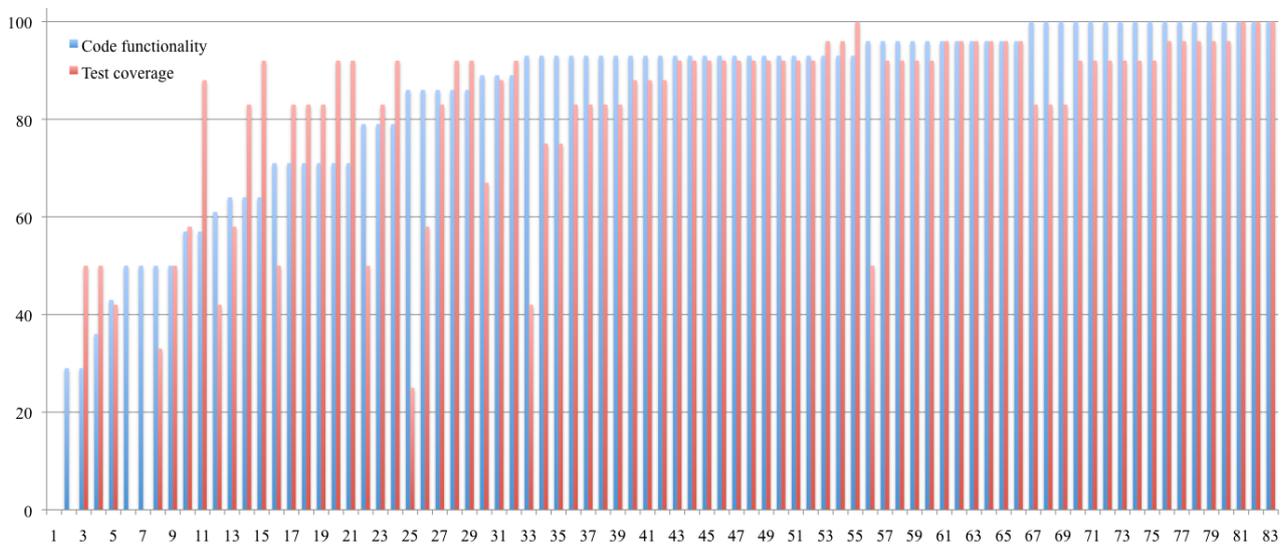


Figure 6: Comparison of marks for code functionality and test coverage, Assignment 1b, 83 submissions

Recall that Assignments 1a and 2a were individual, so we can assume that the same person developed both the unit tests and program code. How pairs of students divided their workload in Assignments 1b and 2b is difficult to say based purely on the submitted artefacts, although if they followed the assignments' instructions they would have swapped 'tester' and 'coder' roles regularly. Figures 6 and 8 are for the pair programming assignments and both show an improvement in the test coverage marks, compared to the preceding individual assignments in Figures 5 and 7. Regardless of how they divided up the task, this suggests that the students took the unit testing parts of the assignment more seriously in their second assessment. Nevertheless, the test coverage results still lag well behind those for code functionality throughout.

Inspection of the submitted assignments suggests that many students *didn't* follow the test-driven development process which, in practice, can be time consuming and requires considerable discipline. Often students developed their program code and their unit tests separately, rather than letting the latter motivate the former. However, even if students did not follow the test-driven development process, this does not invalidate our comparison of their unit testing and program coding abilities, as these are two

distinct skills.

In all four sets of results there are numerous examples of students scoring well for code functionality but very poorly for test coverage, meaning that they could implement a solution to a problem that they couldn't (or merely chose not to) characterise in the form of a unit test suite, the exact opposite of what we would expect if they had strictly followed the test-driven development process.

To explore this phenomenon further we conducted a conventional correlation analysis (Griffiths et al. 1998) to see if there was a clear relationship between students' testing and coding skills, irrespective of their absolute marks for each. We began by checking the normality of all the marks' distributions. This was done by inspecting quantile-quantile plots generated by the *qqnorm* function from the statistics package *R* (R Core Team 2012). These plots (not shown) provide good evidence of normality for seven of the eight data sets, albeit with some overrepresentation of high scores in the coding result distributions. The quantile-quantile plots were skewed somewhat by discretisation of the marks; evidence for normality was especially strong for Assignments 2a and 2b, for which we had the most fine-grained marks available.

The marks were then used to create correlation scatterplots (Griffiths et al. 1998), comparing stan-

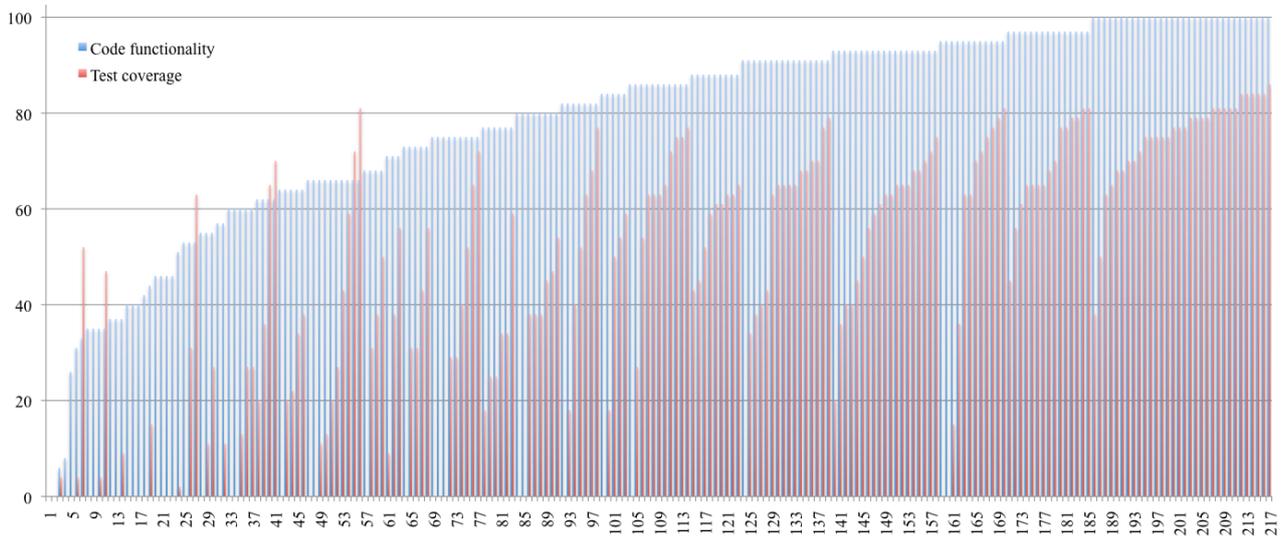


Figure 7: Comparison of marks for code functionality and test coverage, Assignment 2a, 217 submissions

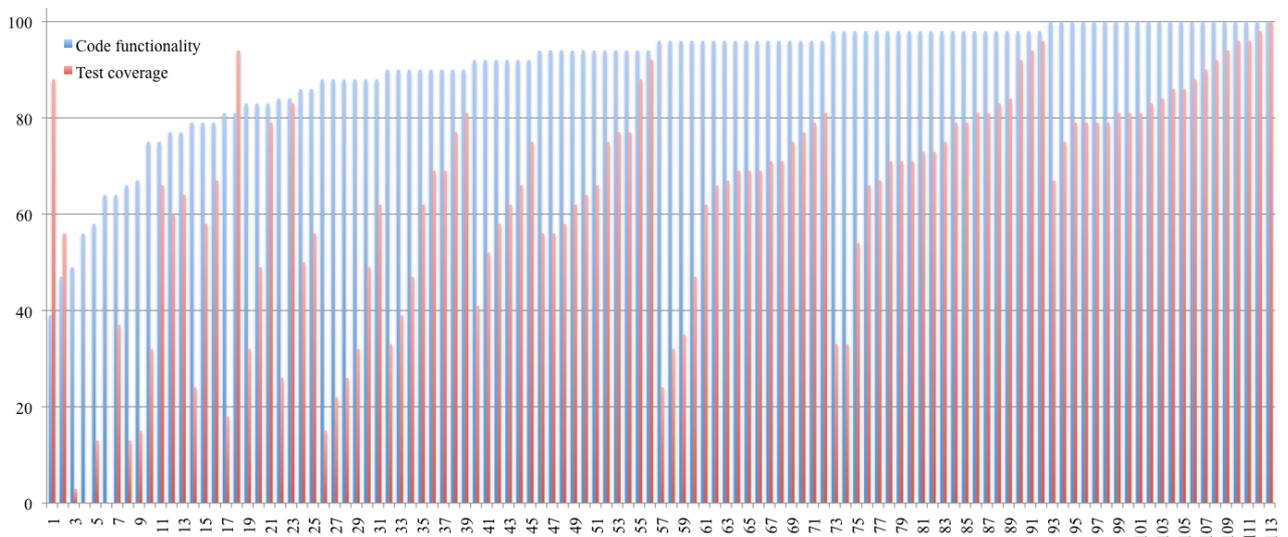


Figure 8: Comparison of marks for code functionality and test coverage, Assignment 2b, 113 submissions

dardised marks for each of the four assignments, as shown in Figures 9 to 12. As usual, dots appearing in the top-right and bottom-left quadrants suggest that there is a positive linear relationship between the variables of interest, in this case students' code functionality and test coverage results. (In the interests of clarity a handful of extreme outliers, resulting from some students receiving near-zero marks for both tests and code, have been omitted from the figures.)

All four figures exhibit good evidence of a linear relationship, especially due to the numbers of students who did well on both measures (top-right quadrant). Overall, though, the pattern is not as strong as we expected.

Undoubtedly many students put less effort into the unit tests, despite them being worth equal marks to the program code. It is clear from the averages in Table 2 and the charts in Figures 7 and 8 that even the best students did not do as well on the unit tests as the program code, and it certainly wasn't the case that successfully defining the test cases to be passed was a pre-requisite to implementing a solution as we had originally assumed. Even more obvious is the line of dots along the bottom of Figure 11 which was caused by students who received zero marks for test coverage. Evidently these students learned the im-

portance of the unit tests for their overall grade by the time they did their second assignment because no such pattern is evident in Figure 12.

Overall, therefore, contrary to our expectations, computer programming students' performance at expressing *what* needs to be done proved to be a poor predictor of their ability to define *how* to do it.

6 Conclusion

The students' poor performance in unit testing compared to program coding surprised us. Nonetheless, the pattern of results in Figures 5 to 8 is remarkably consistent. On the left of each chart are a few examples of students who could characterise the problem to be solved but couldn't complete a solution (i.e., their test coverage results in red are better than their code functionality results in blue), which is what we would expect to see if the students applied test-driven development. However, this was far outweighed in each experiment by the dominance of results in which students produced a good program but achieved poor test coverage (i.e., their blue code functionality results were better than their red test coverage results). We thus have clear empirical evidence that students struggle to a greater extent defining test suites than

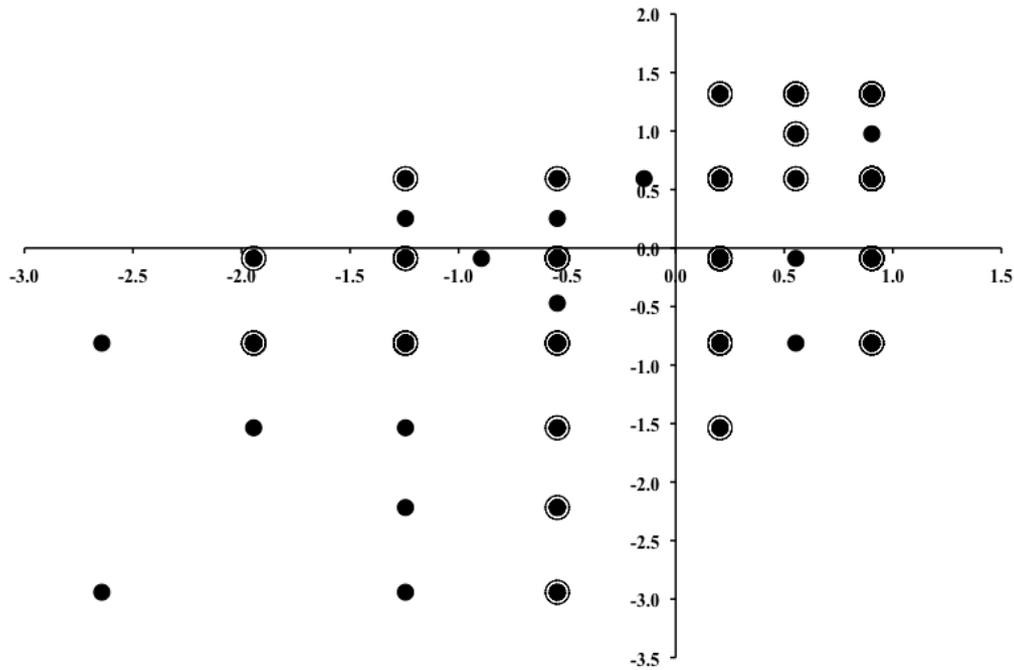


Figure 9: Correlation scatterplot, code functionality (x-axis) versus test coverage (y-axis), with circled dots denoting multiple data points with the same value, Assignment 1a

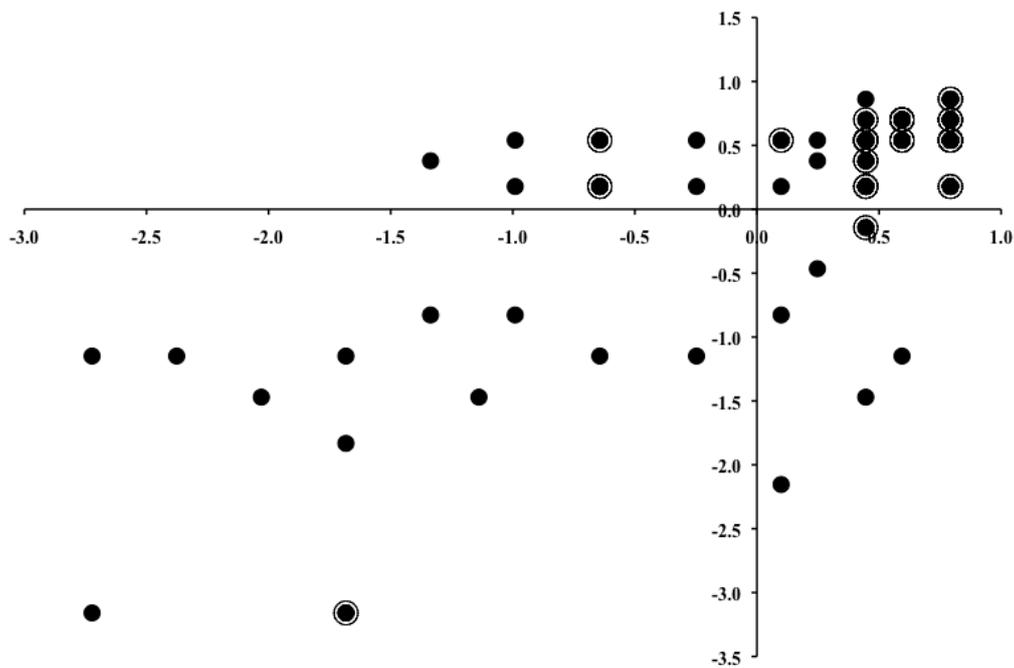


Figure 10: Correlation scatterplot, Assignment 1b

they do implementing solutions, for the same programming problem.

The outstanding question from this study is whether this unexpected result reflects students' abilities or motivations. From our inspection of the submitted assignments we can make the following observations.

- The results can be explained in part by many students' obvious apathy towards the unit testing part of the assignments. Despite their regular exposure to the principles of test-driven development in class, and despite being well aware that half of their marks for the assignment were for their tests, it was clear in many cases that students did not follow the necessary discipline and

instead wrote the program code first, seeing it as more "important". Their unit tests were then completed hastily just before the assignment's deadline. As noted in Section 2 above, this student behaviour has been observed in many prior studies. For instance, Buffardi & Edwards (2012) found that students procrastinate when it comes to unit testing, even when a test-first programming paradigm is advocated, which typically leads to poor test coverage in the submitted assignments.

- Another partial explanation is simply that many students had poor unit testing skills. Test-driven development emphasises the construction of a large number of small and independent tests,

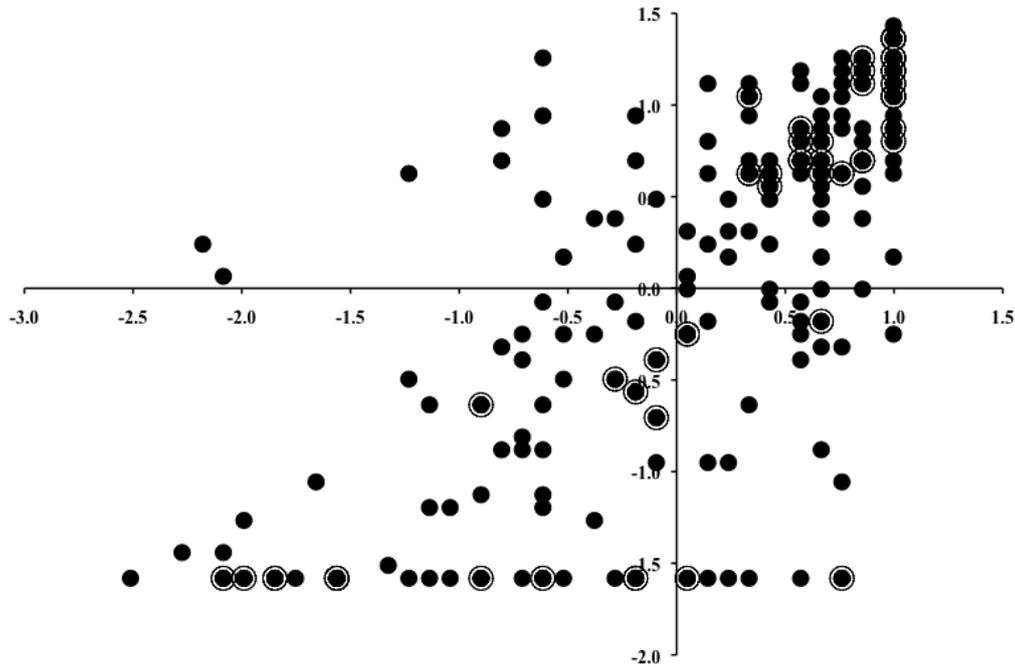


Figure 11: Correlation scatterplot, Assignment 2a

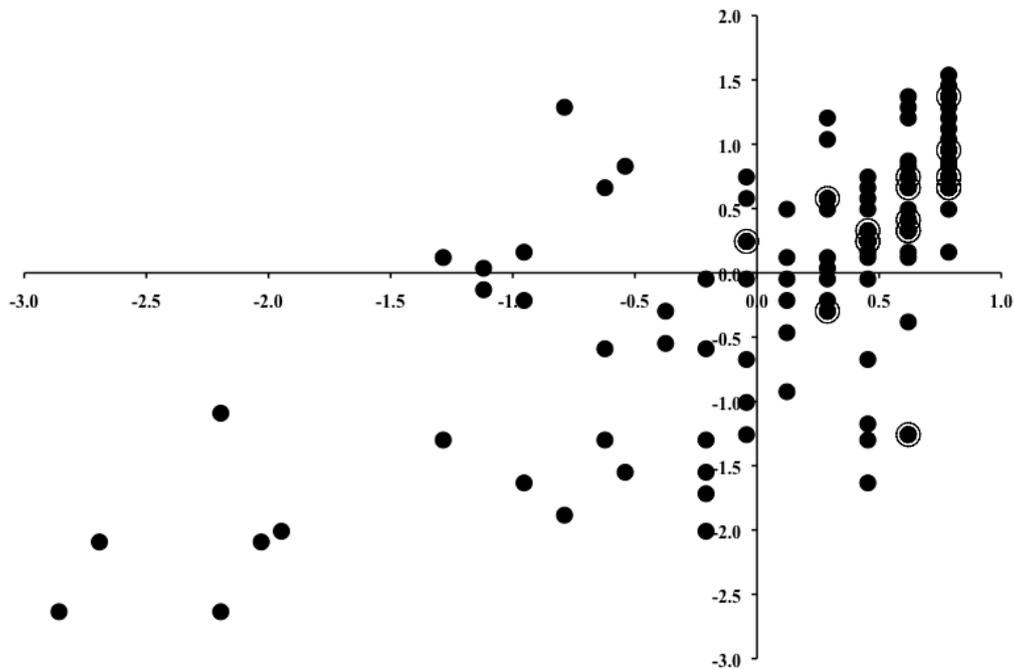


Figure 12: Correlation scatterplot, Assignment 2b

each highlighting a distinct requirement. However, inspection of some assignments with low testing scores showed that they consisted of a small number of large and complicated tests, each attempting to do several things at once. This is evidence that the students could not (or chose not to) follow the test-driven development discipline. Small sets of complex unit tests are characteristic of test-last programming and typically produce poor test coverage because several distinct coding errors may all cause the same test to fail, without the reason for the failure being obvious. At the other extreme there were also a few examples of students producing a very large number of tests, often over twice as many as in our own solution, but still achieving poor coverage because their tests did not check distinct

problems and so many were redundant. This undirected, ‘shotgun’ strategy is again evidence of a failure to apply, or understand, test-driven development, which avoids redundancy by only introducing tests that expose new bugs.

- It is also noteworthy that writing unit tests is a cognitively more abstract activity than writing the corresponding program code, thus making it more challenging for students still coming to grips with the fundamentals of programming. Whether or not this was the cause of students’ poor test coverage marks is impossible to tell from the submitted artefacts alone. Anecdotally, our discussions with students while they were working on the assignments left us with the impression that they understood the prin-

principles of unit testing well enough. By far the most common question asked by students while they were working on their assignment was “How many unit tests should I produce?” rather than “How do I write a unit test?”

Ultimately, therefore, further research is still required. Although we have demonstrated that programming students’ testing and coding skills can be analysed separately, and that they consistently perform better at coding than testing, more work is required to conclusively explain why this is so.

Acknowledgements

We wish to thank Dr Andrew Craik for developing the assignment marking scripts used in the first two experiments, and the anonymous reviewers for their many helpful suggestions for improving the correlation analysis. Support for this project was provided by the Office of Learning and Teaching, an initiative of the Australian Government Department of Industry, Innovation, Science, Research and Tertiary Education. The views expressed in this publication do not necessarily reflect the views of the Office of Learning and Teaching or the Australian Government.

References

- Arisholm, E. & Sjøberg, D. (2004), ‘Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software’, *IEEE Transactions on Software Engineering* **30**(8), 521–534.
- Astels, D. (2003), *Test-Driven Development: A Practical Guide*, Prentice-Hall.
- Barriocanal, E., Urbán, M.-A., Cuevas, I. & Pérez, P. (2002), ‘An experience in integrating automated unit testing practices in an introductory programming course’, *SIGCSE Bulletin* **34**(4), 125–128.
- Beck, K. (2003), *Test-Driven Development: By Example*, Addison-Wesley.
- Buffardi, K. & Edwards, S. (2012), Exploring influences on student adherence to test-driven development, in T. Lapidot, J. Gal-Ezer, M. Caspersen & O. Hazzan, eds, ‘Proceedings of the Seventeenth Conference on Innovation Technology in Computer Science Education (ITiCSE’12), Israel, July 3–5’, ACM, pp. 105–110.
- Desai, C., Janzen, D. & Savage, K. (2008), ‘A survey of evidence for test-driven development in academia’, *SIGCSE Bulletin* **40**(2), 97–101.
- Edwards, S. (2004), Using software testing to move students from trial-and-error to reflection-in-action, in D. Joyce, D. Knox, W. Dann & T. Naps, eds, ‘Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE’04), USA, March 3–7’, ACM, pp. 26–30.
- Ferguson, T. S. (2010), ‘Optimal stopping and applications’. <http://www.math.ucla.edu/~tom/Stopping/Contents>.
- Griffiths, D., Stirling, W. D. & Weldon, K. L. (1998), *Understanding Data: Principles and Practice of Statistics*, Wiley.
- Janzen, D. & Saiedian, H. (2007), A leveled examination of test-driven development acceptance, in ‘Proceedings of the 29th International Conference on Software Engineering (ICSE’07), USA, May 20–26’, IEEE Computer Society, pp. 719–722.
- Janzen, D. & Saiedian, H. (2008), Test-driven learning in early programming courses, in S. Fitzgerald & M. Guzdial, eds, ‘Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE’08), USA, March 12–15’, ACM, pp. 532–536.
- Keefe, K., Sheard, J. & Dick, M. (2006), Adopting XP practices for teaching object oriented programming, in D. Tolhurst & S. Mann, eds, ‘Proceedings of the Eighth Australasian Computing Education Conference (ACE2006), Hobart’, Vol. 52 of *Conferences in Research in Practice in Information Technology*, Australian Computer Society, pp. 91–100.
- Link, J. (2003), *Unit Testing in Java*, Morgan Kaufmann.
- Melnick, G. & Maurer, F. (2005), A cross-program investigation of students perceptions of agile methods, in ‘Proceedings of the 27th International Conference on Software Engineering (ICSE05), USA, May 15–21’, ACM, pp. 481–488.
- R Core Team (2012), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.
URL: <http://www.R-project.org>
- Schach, S. (2005), *Object-Oriented and Classical Software Engineering*, McGraw-Hill, USA. Sixth edition.
- Schuh, P. (2005), *Integrating Agile Development in the Real World*, Thomson.
- Spacco, J. & Pugh, W. (2006), Helping students appreciate Test-Driven Development (TDD), in W. Cook, R. Biddle & R. Gabriel, eds, ‘Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA06), USA, October 22–26’, ACM, pp. 907–913.
- Whalley, J. & Philpott, A. (2011), A unit testing approach to building novice programmers skills and confidence, in J. Hamer & M. de Raadt, eds, ‘Proceedings of the Thirteenth Australasian Computer Education Conference (ACE 2011), Perth’, Vol. 114 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society, pp. 113–118.

Visualisation of Learning Management System Usage for Detecting Student Behaviour Patterns

Thomas Haig

Katrina Falkner

Nickolas Falkner

School of Computer Science
The University of Adelaide
North Terrace, South Australia

Email: [thomas.haig,katrina.falkner,nickolas.falkner]@adelaide.edu.au

Abstract

Identifying “at-risk” students - those that are in danger of failing or not completing a course - is a crucial element in enabling students to achieve their full potential. However, with large class sizes and growing academic workloads, it is becoming increasingly difficult to identify students who require urgent and timely assistance. Efficient and easy to use tools are needed to assist academics in locating these students at early stages within their courses. A significant body of work exists in the use of student activity data, e.g. attendance, performance, participation in face-to-face and online sessions, to predict overall student performance and at-risk status. This is often built upon the considerable amount of student data within learning management systems. Manual data collection, including surveys and observation, which introduces additional workload is often required to extract relevant data meaning that it in large classes it is prohibitively difficult to apply such techniques.

In this paper, we introduce a framework for at-risk identification combining simple metrics, gathered from social network and statistical analysis domains, that have been shown to correlate with student performance and require slow amounts of manual data collection or additional expert analysis. We describe each of the metrics within our framework and demonstrate their usage. We use visualisation to enable easy interpretation of results. The application of our framework is demonstrated within the context of an advanced undergraduate computer science course.

Keywords: Student data, Learning Management Systems, Prediction, Visualisation

1 Introduction

In order to enable all of our students to succeed to their potential, academics must be able to identify students who are “at-risk” - those students who are likely to fail, or withdraw, from a course - within the early stages of their at-risk behaviour. Interventions can only be made if academics have efficient and clear facilities that enable them to identify at-risk students. At-risk behaviours are becoming increasingly difficult to detect within our overburdened higher education systems, with large classes, de-personalised administrative systems and separation from peer groups. Our

classrooms are increasing in diversity (Biggs & Tang 2007), further complicating this issue by presenting us with an “unprecedentedly broad spectrum of student ability and background” (Ramsden 2003).

Early identification of at-risk students is of particular concern within the ICT discipline - within Australia, and globally, we have seen a recent dramatic drop in applications for ICT degree programs, poor progression and retention rates (Sheard et al. 2008).

In order to identify at-risk students we need to provide facilities to assist academics in finding these students. Any facilities or tools provided to assist academics must introduce minimal additional workload. Although true in every discipline, Computer Science and ICT academics face an increasing pressure to include more technical concepts in their curriculum (McGregor et al. 2000). In 1978 the ACM recommendations for undergraduate programs consisted of a 20-page document. In 2010, the current recommendations total 240 pages with a vast increase in the body of knowledge expected of an undergraduate curriculum (Becker 2008). These pressures, along with pressure from industry and accrediting bodies to focus more attention on the development of generic skills (Falkner & Falkner 2012), mean that ICT academics must find efficient and effective mechanisms to assist them in these tasks. Further, we must work within the available data sources that can be readily accessed by academics within their institutions.

There has been considerable work within the area of automated at-risk identification, using a variety of data sources, such as learning management systems, grade rosters, attendance records, and participation in online discussion forums. However, the majority of this work relies upon a blend of automated analysis and manual coding or recording of data. This includes the use of surveys and large-scale data collection to complement automatically available data. Even within a small cohort, these methods present an additional workload for academics, which becomes prohibitive within large classes, where these techniques are often most needed.

In this paper, we propose a framework for the identification of at-risk students using a combination of simple metrics, gathered from the domains of social network analysis and statistical analysis. These metrics, based upon data readily available within learning management systems, present an automated analysis framework requiring minimal manual interaction with the underlying data, and no additional data collection. We present a range of data visualisations that enable academics to easily and efficiently identify students who are exhibiting potential at-risk behaviours. We are able to gather the required data early on in a course without the requirement for additional assessments or surveys.

Copyright ©2013, Australian Computer Society, Inc. This paper appeared at the Fifteenth Australasian Computing Education Conference (ACE2013), Adelaide, Australia, January 2013. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 132, Angela Carbone and Jacqueline Whalley, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

Using our obtained data and visualisations, we aim to find patterns of behaviour for successful and unsuccessful students for specific activities of a particular course. By identifying patterns of behaviour we may be able to identify those students who require additional assistance and intervention. We present visualisations for each metric that support the ease of identification of such patterns, and hence the subsequent identification of at-risk students.

We demonstrate the utility of our framework within a case study of an advanced undergraduate ICT course, using data available from the learning management system used within the course, the Moodle Learning Management System, which is used by multiple institutions around the world (Moodle 2007). We demonstrate each of the metrics within the framework applied to this case study, exploring visualisation potential and validating correlation for each metric and student performance.

The paper will be presented as follows. Section 1 presents an Introduction, Section 2 a review of related work, Section 3 our Methodology, Sections 4, 5 and 6 our Results, Section 7 a Discussion and Section 8 will present potential future work and conclusions.

2 Related Work

Obtaining measures of student engagement provides us with one method for determining at risk behaviour. However, traditional methods of ascertaining student engagement, such as attendance and participation in lectures and tutorials, become increasingly difficult to use in large classes. With the increasing conflict between external pressures, such as work and family commitments, even engaged students may not be always able to make frequent face-to-face contact. Further, there is a time burden involved with taking these attendances and measuring participation, which academics may struggle to afford. We want to assess engagement automatically in order to predict results for students, so as that we may find those who may be at risk of failing a course, or dropping out, in order to attempt to prevent this occurring.

The increasing use of online learning management systems and online learning tools means that we now have alternative means for gathering information on student engagement, which supports the more flexible practices of the modern higher education sector.

Large amounts of data are readily available for analysis of student engagement. Bayer et al. (2012) have explored the use of a wide variety of data including “capacity to study” test scores, attained credits, average grades, gender and year of birth, to develop a model of social behaviour in order to predict potential drop-outs. Merceron & Yacef (2005) utilised course specific information, such as the types of mistakes made in individual assessment exercises, while El-Halees (2009) used preliminary course assessment results as a prediction of final grade, hence identifying at-risk students.

Norris et al. (2008) looked at the work of novice programmers in the BlueJay environment. This allowed them to log specific actions such as number of compilations made, amount of time spent working on a project and the amount of errors encountered. This level of data provided a log of students patterns of work while performing a programming task in a short closed session. Their aim was to gather patterns of work for successful students and compare these with unsuccessful students visually and intervene where necessary to get unsuccessful students into better patterns of work, for example compiling their

code more often. While we are not able to log results at this level, it is useful to note that patterns of behaviour run to deep levels within a course, down to how students portion their time and work while programming.

Logging at a similar level is performed by Murphy et al. (2009) who also use a BlueJay or Eclipse plug-in to explore student programming habits. They logged the students time spent on assignment as well as their compilation errors. They then used this data to send recommendations to students about how they could improve, e.g. they are spending too long on an assignment, or making the same error too many times. Students are then able to reflect upon their patterns and this aims to move students towards more successful patterns of study, through self-intervention. Instructors were also able to use the data to make more meaningful interventions, specifically where students were making numerous errors it was shown that early intervention was able to help.

The work of Norris et al. (2008) was expanded upon by Fenwick Jr et al. (2009). Using the same ClockIt software, Norris et al. (2008) observed patterns of student behaviour in a programming task and how this potentially relates to cheating, as well as how much incremental work they put into their programming. They found that students that started the task later in general received a lower grade. As noted in their work “although this is what we have already been “preaching” to students, it is now based on objective analysis of quantitative data”.

Edwards et al. (2009) analysed submission data and came to the same conclusions as Norris et al. (2008), in that students who start their work early, in their case as measured by their first submission, are more likely to perform better in a task. Of note from their study is that students who perform consistently well or consistently poorly may demonstrate similar behaviours, e.g. a good student may start consistently late and be able to perform under pressure.

Nandi et al. (2011) take the online participation of students in forums as a measurement of engagement and a possible predictor of grade. Their results show that students who input more into the course achieve a better grade. The course analysed was fully online, and hence students point of contact to course providers was through the forums, hence overall forum usage was high. This is in contrast to a blended learning course where students have more access to course providers without using the online environment, and hence overall participation is lower.

Tracking of student movement through an online learning website was performed by Ceddia et al. (2007) using web logs. They use these logs to track student behaviour and categorise it as either purposeful or browsing behaviour. They found that as the course progressed students use the online system more purposefully, browsing less to get to their required goals and materials. They also used the logs to analyse the learning behaviours of students on the website. They used completion rates, duration, frequency to measure effectiveness, efficiency and explorational activities. They also used unusual results to find possible problems with the website interface.

Students self-managing their own timesheets was shown by Herbert & Wang (2007) to be an effective measure of students usage of an online learning system. Students self-evaluated their use of the system, and this was then contrasted with the actual usage data from the website. They sought to find behavioural patterns that showed students may work to deadlines, relate their time spent to marks available and to test if students could be induced to

start tasks early. They found student timesheets accurate enough to be able to critically analyse these behaviours, backing up anecdotal evidence that students do indeed work to deadlines and will only spend as much time as proportional to marks.

Although successful in identifying at risk potential, these studies utilise data that is not readily available across the sector, or not readily available in a timely fashion to perform early intervention.

Studies such as Sheard et al. (2003) and Georg (2009) supplement their automatically collected data with manually collected survey data. Sheard et al. (2003) utilised survey data to gain student ratings of how useful online course material is to them, and how useful the online site is to their studies. Georg (2009) used the Konstanz Student Survey, collected in Germany every two to three years, which collects data about students attitudes towards study and profession to analyse factors that lead to students dropping out of courses. However, the use of surveys is problematic. Black et al. (2008) suggest the burden of time on administration to create surveys is obvious and, further, students already suffer from “survey fatigue”, where over-surveying causes data to become skewed due to students aiming to simply complete surveys, not give objective answers.

Data collection may be followed by a phase of addition, where the data is inspected manually and supplemented with expert evaluations. This is used in studies such as Lopez et al. (2012), where forum posts are viewed and analysed by experts within the field in order to evaluate their worth. This expert rating is then added to the data as a measure of how useful a resource will be to a student, and hence how much potential benefit they may receive from using it. This re-analysis of data, after its collection, is a further demand on academics time as the manual inspection of data is extremely time consuming. We would also argue that students are able to be their own “expert evaluators” of the data that is the most relevant to them. They are able to identify resources that are useful and hence successful students will access and use these resources more frequently.

Visualisation of a network in a learning management system has been carried out in studies such as Dawson et al. (2010). Dawson et al. (2010) showed the structure of the social network created between students when they interact on the forums in a learning management system. The use of student patterns as predictors of final result has been studied in Zhang et al. (2007) and Casey et al. (2010) however they do not present their results visually and give their results in a more “raw” format, which requires a degree of statistical knowledge and insight to understand and work with.

3 Methodology

We aim to create an “at-risk identification” framework by combining simple, automated metrics and simple visualisations for assessing student behaviours. We gather methods from a range of areas, including social network analysis, statistical analysis and data visualisation.

Studies such as Zhang et al. (2007) and Casey et al. (2010) show that successful students are more frequently and regularly participating and engaged in online activities. Much of the work within the area of at-risk identification addresses the early identification of students that exhibit conflicting behaviours, i.e. they are not engaged in the course and are not actively participating. One of the most accurate mea-

asures of student engagement is student performance in assessment activities, but this may not promote timely interventions, as assignment work may come too late in a course. We utilise data available in learning management systems, such as access to online course resources and participation in forums, as measures of engagement.

Measures such as frequency of access are somewhat coarse and require more detailed analysis to determine engagement. A student may be accessing many resources but they may not be relevant to the activities currently at hand, or alternatively a student may be accessing only a small number of resources, but those that are the most directly relevant to their work. We would argue that the latter student is the more successfully engaged in their studies, and hence measures of the frequency of accesses only presents a partial picture, and may incorrectly categorise students.

Accordingly, we propose a framework that tracks students activities over time, combined with their frequency of accesses.

We utilise three distinct methods to explore student engagement:

- Social Network Analysis (SNA) - SNA techniques enable us to explore relationships within our “network”, which consists of data contained within the LMS, such as forum postings and course resources, the student cohort, and connections from students to the data, i.e. a student may read a forum message posted by another student.
- Frequency of Access Analysis - analysing patterns of access for individual resources and student access patterns over time.
- Measure of Distance Analysis - analysing patterns of access behaviour and similarity between student access patterns both visually and quantitatively.

Using the combination of these metrics, we are able to identify patterns of behaviour based upon participation in online course activities. We have developed visualisations for each approach that can be used by academics to identify students who are demonstrating patterns of at-risk behaviour within the context of their course.

We discuss a students success based upon their grades received in the course. There are five grades awarded:

- High Distinction - A grade of or over 85%. Due to our small sample size we only had one High Distinction student in the course, as such their result has been put with the next grade band down, to form a larger “Distinction” grade band of students.
- Distinction - A grade of or over 75%.
- Credit - A grade of or over 65%.
- Pass - A grade of or over 50%. This is the lowest acceptable passing grade.
- Fail - A grade below 50%. This is the only course failure grade.

3.1 Social Network Analysis

In order to show the data visually we use Social Network Analysis (SNA) techniques such as in Dawson et al. (2010).

We use SNA to give a simple display of the network of students participating in a course. The forum network on an LMS can be considered to be a social network, where students “socialise” by accessing the same materials or interacting asynchronously on the forums. Analysing this network allows us to see the extent with which students are engaging with each other and the course materials.

SNA creates a network made up of a set of “actors” who have a relationship with each other. The actors in the course are students who interact with each other via resources. They are related when two of the same actors have accessed the same resource. It is not necessary for all actors in the network to be related, as both present and absent connections are taken into account. SNA is used to explain the network of actors and the effect of the relationships in the network. In our case we can use absent connections to find those who are disengaged from the network and hence likely to be disengaged from the course.

3.2 Statistical Measures

We will analyse student engagement by measuring student access to the LMS in two ways, access to individual resources and frequency of access to resources. We believe that students who are more engaged with a course will seek to access a wide array of materials and that they will access them frequently as required. We shall count distinct accesses to resources as the number of times a resource has been accessed and a date access as days on which students actively participated with the LMS, by accessing materials or engaging on the forum.

We begin by looking at binary-yes-or-no access counts, with a resource having been accessed or an access occurring on a date. We then use a box-and-whisker plot to show the differences between these accesses frequencies in comparison to other students by grade band. When presented with data from a new student an academic would be able to check their accesses frequencies and give a high level assessment of their engagement. For a more fine-grained analysis we will then look at the distinct number of accesses made to each resource and the number of accesses made on a given day. We will visualise these results using a “heat map”, which shows gradients of access frequencies. The heat map shows, using colour gradients, the intensity of activity on this resource or on a particular date. The darker the colour on the heat map the more frequent the activity occurring, i.e. darker points indicate more intense activity. From this we expect to see patterns of student behaviour that we can compare with other students or new data.

After a course has been run we are able to create an averaged pattern of accesses, i.e. what action the majority of students have taken. This may be useful after multiple iterations of the same course as an academic could compare new data to the “average” pattern.

3.3 Implementation

We have implemented our framework using data from the Moodle LMS. Moodle logs a large amount of student participation data in a CSV file, which is readily

available in all Moodle installations. The data contains four entries for each “action” performed on the system, these are:

- **Full Name** of the student or lecturer accessing the system.
- **Date** on which the action occurred.
- **Access Type**, a shorthand for the type of action performed.
- **Information**, the name of the resource on which the action was performed.

The set of actions that can be taken on the system is extensive and we shall not cover all possible actions here. The set of actions are broken into five categories of access; User, Course, Resource, Administrative and Forum. Of relevance to us for this research are Resource and Forum accesses, of which all possible actions are:

- **Resource Access**

Resource View: view a course resource. The information then contains the name of this resource.

- **Forum Access**

Forum View Forums: View all available forums, i.e. a list of sub-forums if such forums exist.

Forum View Forum: View all discussion headings in a specific forum.

Forum View Discussion: View a particular discussion on the forum, also included is the title of the forum.

Forum Add Post: Add a new post to a discussion forum.

Forum Update Post: Update a post that was previously created.

Relevant Moodle data from the system is imported into an external MySQL database to support queries. MySQL is a ubiquitous and free resource that is easily installed and is the most popular open source database system in the world (MySQL 2012). The database also contains grade information for the students.

The SQL database is then integrated with a Python program which allows us to extract and process the data. Python is a free, ubiquitous open source product (Python 2012). The output from the Python program is stored which allows the visualisations to be re-run as required without a large amount of space overhead.

The file produced from the Python program is then read into R. R is a statistical program that features many built in libraries capable of helping to visualise the data, as well as run statistical analysis and once again is free and open source (R Core Team 2012). We use R to produce the final visualisations which we use to identify patterns in the data.

The outlined process of creating these visualisations is able to be automated using scripts, which run each of the required programs in order, negating the need to perform all of these steps manually.

The data which we have used is from a typical third year course run at the University of Adelaide. The data represents a course that contained 47 enrolled students, with 44 completing the course and receiving a final grade, and has 22,320 unique data entries logged on the Moodle system for this course.

The spread of student grades were from a high score of 90, down to a low of 38 out of 100. We experimented on data from another course to check the validity of our results and found our results to be typical of the courses under study. We present data from a single course for clarity.

4 Network Visualization

We aim to show the network as a whole and show key points of the network, such as its density and make-up. A view of how grades are spread in the network relative to the density or sparsity of links between students will show if there are any distinct communities of grades within our network.

Students are categorized as having a “link” in the network if they have read a discussion that has been started by another student. We take all relations to be didactic, as directionality is not important, students who are engaging in the course will be represented as having a link with other students accessing the same materials.

In Figure 1 we see the visualization of the network as a whole. This gives us a view of the engagement with the course by showing students linking with the materials.

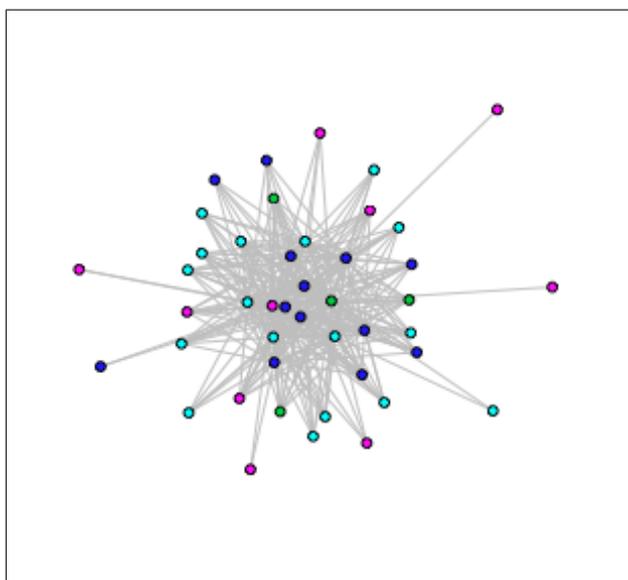


Figure 1: Visualization of the overall Network of students.

We see an incredibly dense network, even for this very small cohort. We see that students on the outer edges of the network are those that are less engaged with the course, and are more likely to fail. This behaviour is supported by similar analysis in Dawson et al. (2010). This gives a good initial guide into the overall structure of the network, and is a quick method to identify students who are potentially at risk as labels are able to be retrieved.

By using this network analysis, educators are able to get a “snapshot” of their class as a whole, potentially allowing them to target students on the fringes of the network for assistance and to get them re-engaged with the course.

From this we can pick out the students who are not well connected to the network. These are the students for whom intervention is needed, as they are not accessing materials. This can be monitored before any assignments have been marked and requires no additional assessment or materials. Further, we can view

changes over time to see if students who have been encouraged to interact begin to do so. We cannot, however, see when students have become disengaged who were engaged previously. These students may begin to drift more towards the fringe of the network, but the drift will be slow and likely not noticeable, hence we need different metrics to identify this type of case.

5 Frequency of Access

The materials with which a student engages, and the frequency with which they access them, are more informative than simple measures of overall engagement. Due to the large amount of resources in any one course it may be easy for students to access irrelevant materials, however students who access relevant materials, at the correct time, are those who are most likely to succeed. As such we look at measures of access frequencies and patterns, exploring the behaviours of both successful students and those who fail the course. We shall look at frequency of access in two ways, both of which are engagement through examination:

- By resource, where it is shown how many course resources a student has viewed.
- By date, where it is shown how many access to resources on a particular date are made.

We firstly look at the raw counts of student accesses and compare these in grade bands using box-and-whisker plots. Within these plots our grade bands are numbered, specifically:

- 1 - High Distinction and Distinction grade students.
- 2 - Credit grade students.
- 3 - Pass grade students.
- 4 - Failing grade students.

Hence, the first three boxes all refer to passing grades, while the fourth refers to a failing grade.

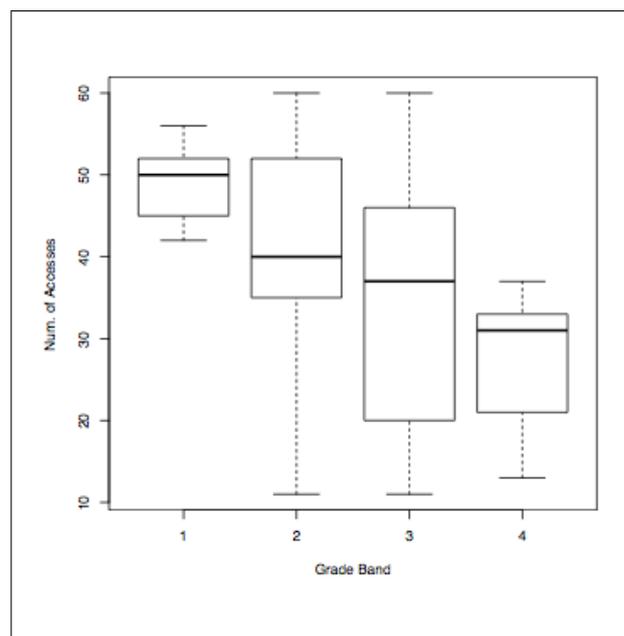


Figure 2: Box and Whisker plot of results by Resource

Figure 2 provides us with a box-and-whisker plot of resource access counts and shows that students in the highest grade bands have a greater mean number of resource accesses, accessing around 50 distinct resources. Credit students have a mean of around 40, similar to that of passing students, however have a much smaller Inter-Quartile Range, or dispersion between students, showing that Credit students on average have a higher access rate than passing students. Finally, failing students have the lowest mean number of resource accesses, with around 30. The bound on the 75th percentile for failing students falls below the mean for passing students (in fact the longest whisker of the failing students is still below the mean for passing students). This shows a very evident difference in behaviour and that there is correlation between frequency of access and a students final grade.

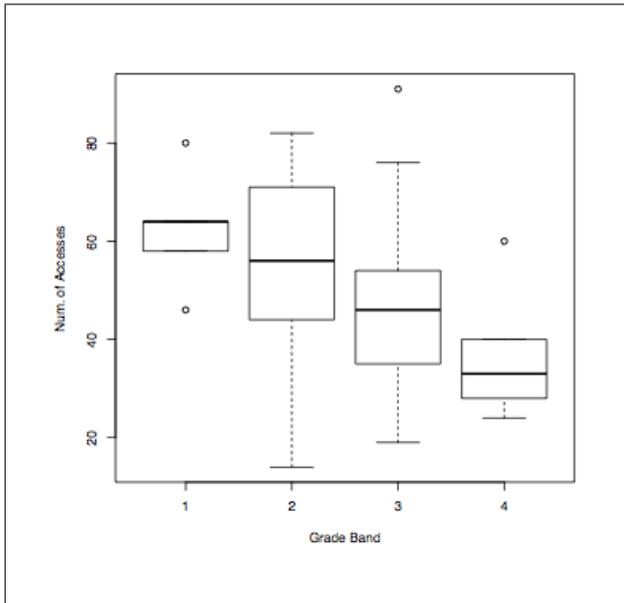


Figure 3: Box and Whisker plot of results by Date

In Figure 3 we look at accesses by date. When viewing by date similar patterns to that of by resource are evident. Again the means increase with grade. Credit students have a large Inter-Quartile Range meaning that their accesses varied, however the mean was still greater than that for passing students. What is clearly evident, is that lower achieving students have a lower rate of access than higher achieving students. Failing students access the forums on around 30 days of a course that ran for over 100 days, in comparison to a mean of 45+ dates for passing students and 60 for Distinction students.

From this we can check an individual students access levels against the rest of the cohort. If their accesses are in the lower ranges then there is a possibility that they may be at-risk. The box and whisker plots give a raw numerical output and do not show more timely accesses or accesses to more important resources.

To achieve a more fine-grained view of student accesses we observe, on an individual level, the engagement with the forums and find if this has a bearing upon the final grade of a student. We visualise our results using a heat map which allows us to observe broad student access patterns, including how frequently students return to view a resource. Key resources may be accessed more frequently, for example a forum post with a long, relevant discussion. In order to show the results clearly we limit the number

of “return” events to nine. A student did access some resources 50+ times, however this is considered an outlier and would make it difficult to see lower number of accesses. It would be expected that students that have a higher rate of access are more engaged with the course.

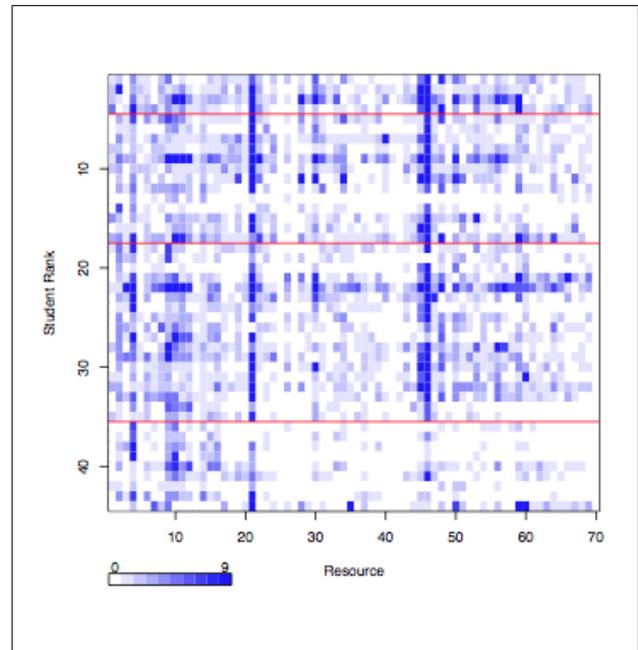


Figure 4: Results of Frequency of Access by Resource. Results are ordered by final grade from top to bottom. Resources are ordered in date of posting.

In our figures we show the edges of the grade bandings with heavy horizontal lines, which are ranked from Distinction to Credit, Pass and Fail. Figure 4 gives us evidence that lower grade students have more sparse access levels than high grade students. The lower access rate for failing students is indeed very clear, hence if we were presented with a student with a similarly low level of engagement we would expect them to obtain a similar result. Similarly, we can see if students are not accessing a key resource as frequently as their peers. For example resource “46” has been accessed multiple times by students in the highest grade bands, however the same result is not seen for students who go on to fail the course. This type of information is critical when evaluating student performance against their peers, and gives likely indicators of why their results may be unsatisfactory.

We now look at the frequency of access by date in Figure 5. Again, we would expect that students who are more successful would return to the forums more frequently over a larger range of days. More over it is expected that they access the materials at relevant times, much like accessing relevant resources.

The matrix clearly shows periods of more intense access and that students with a higher grade are accessing the forums more frequently during these periods. It shows a “ramping up” of accesses at certain periods that correspond to when assignments are due as well as the date of the final examination. As an example, between days 70 and 80, high-achieving students access the forums almost daily while failing students have a considerably lower amount of accesses. Overall we can see that failing students have a lower rate of access.

When presented with a new student an academic can check their raw access counts first, which will give

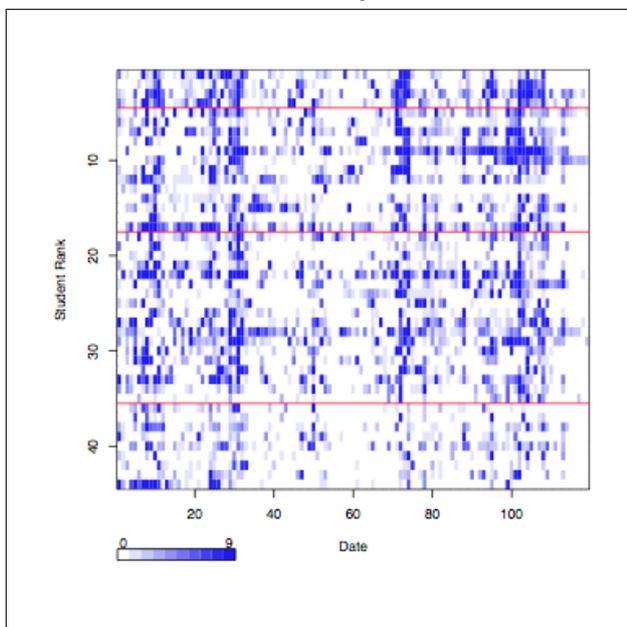


Figure 5: Results of Frequency of Access by Date. Results are ordered by final grade from top to bottom. Dates are from the first day of the course to the end of the examination period.

a good guide as to a student's possible result, and then obtain more accurate detail using the heat maps.

6 Distance Measures

Finally, we begin grouping students into grade bands to find if there are particular behavioural patterns over each grade band. We have shown previously that there is correlation between behaviour and the student grade result, we now aim to quantify these differences and examine the behaviour of each grade band. This would be most useful after multiple cohorts have run through the same materials, allowing quick comparisons to multiple years worth of student data. It also shows the relative “distances” between grade bands of students. When presented with a new piece of data an academic could compare it to the averaged results of each grade band, and find which band the new student is closest to, which would be the likely predictor of their grade.

We obtain the distances between students or bands of students as a Hamming Distance, where we calculate the difference in magnitude of two binary “behavioural vectors”. These behavioural vectors are obtained by filling a vector with a binary 1 or 0 related to the behaviour, with a 1 either being an access to a particular resource or an access on a particular date, a 0 is the absence of this behaviour. Hamming Distance finds the difference in characteristics between two vectors. The Hamming Distance is equivalent to the count of 1s in $s_i \text{ XOR } s_j$. A smaller Hamming Distance will be found for bands with similar access patterns and a large distance for bands with different behaviour patterns. We use a heat map to visualize these differences. For the grade bands we take the average behaviour for that grade band, e.g. if two of three students access a resource it is counted as accessed, if one of three accesses it, it is not. The Hamming Distance is calculated on these averages to find the differences between grade bands. A darker colour indicates a greater similarity, and grade bands are labelled as earlier.

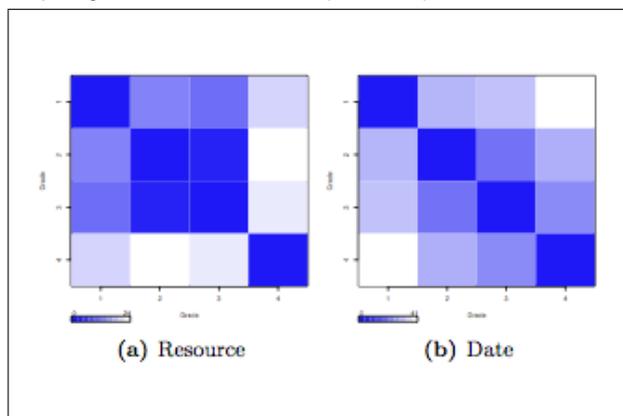


Figure 6: Results grouping by grade banding. Grades run from Distinction to Fail, top to bottom, left to right.

When taking the results averaged over grade band in Figure 6(a) we see differences between high grade and failing students. When looking at resource accesses we see that:

- Credit and pass students are most similar
- Credit and pass students are more similar to distinction students than to failing students.
- Failing students behave differently to all other grades.

When looking at accesses by date in Figure 6(b) we see similar results:

- Failing students are very different to Distinction students.
- Credit and Pass students are still most similar
- Credit students are almost as similar to Distinction students as to Passing students.

From this, when presented with a new student we can combine these two metrics to get the best predictor of their likely result. There may be some difficulty separating Credit and Pass students, however we are, in a large number of cases, able to separate students who are failing from those who are passing the course.

7 Discussion

We have created a framework to assist academics in assessing whether a student is at risk, without imposing additional burdens in time and administration. We have shown that surveys or the manual inspection of data may not be necessary as we were able to achieve useful results using data easily obtained from a commonly used LMS.

We find that a student's pattern of study, as measured by LMS usage, correlates with their final grade in a course. Our results are supported by studies such as Lopez et al. (2012) and Morris et al. (2005), who obtained similar results.

Students who are more engaged with the course perform better in terms of their final grade. This result is not unexpected, however, finding a way to measure engagement and participation is difficult with larger class sizes and current administrative burden. Our framework will provide a method with which academics can measure and visualise participation more

easily. Further, we also find that students who perform better in the course access the most relevant materials at the appropriate times. This is shown by Credit students accessing the same materials as Passing students, but doing so on more appropriate dates.

We present the data obtained visually, allowing it to be more widely understood, including by those without a strong statistical background. This allows academics to identify students who are at risk of failure, if they are found to be demonstrating patterns that correlate to this outcome. We have created this resource without the need for a large amount of manual overhead on the part of the academic and provided them with a simple medium with which to make predictions.

We use this data to guide the creation of an automatic system for detection of failing students, providing an indicator of the types of flags that may be put up as warning signs that students are at-risk.

8 Conclusion and Future Work

We have shown that LMS usage can be correlated with grade, and that we can use simple, highly automated metrics and visualisations to capture students who are potentially at-risk. By doing this we hope that we can increase retention rates as well as course performance and overall facilitate better learning outcomes for students. The system provides an automated tool in order to create these results.

There is the potential for this type of tool to be built directly into Moodle, as shown with SNAPP (Dawson et al. 2010) decreasing the amount of burden to the educator. Further, we can increase automation by checking correlations automatically and creating alerts for course co-ordinators to contact students when necessary, rather than the educator performing this step manually.

Overall, we have shown we are able to find patterns for successful, as well as at risk students, and use these to make predictions about likely outcomes. Doing so may be a step toward decreasing failure rates and increasing retention rates.

References

- Bayer, J., Bydzovská, H., Géryk, J., Obšivac, T. & Popelnský, L. (2012), Predicting drop-out from social behaviour of students, *in* 'Proceedings of the 5th International Conference on Educational Data Mining-EDM 2012', pp. 103–109.
- Becker, K. (2008), 'The use of unfamiliar words: writing and cs education', *Journal of Computing Science in Colleges* **24**(2), 13–19.
- Biggs, J. & Tang, C. (2007), *Teaching for Quality Learning at University, 3rd edition*, The Society for Research into Higher Education.
- Black, E., Dawson, K. & Priem, J. (2008), 'Data for free: Using lms activity logs to measure community in online courses', *The Internet and Higher Education* **11**(2), 65–70.
- Casey, K., Gibson, P. & Paris, I. (2010), 'Mining moodle to understand student behaviour', *International Conference on Engaging Pedagogy 2010 (ICEP10)*, National University of Ireland Maynooth.
- Ceddia, J., Sheard, J. & Tibbey, G. (2007), Wat: a tool for classifying learning activities from a log file, *in* 'Proceedings of the ninth Australasian conference on Computing education-Volume 66', Australian Computer Society, Inc., pp. 11–17.
- Dawson, S., Bakharia, A. & Heathcote, E. (2010), Snapp: Realising the affordances of real-time sna within networked learning environments, *in* 'Proceedings of the 7th international conference on networked learning, Aalborg 3-4th May'.
- Edwards, S., Snyder, J., Pérez-Quiñones, M., Allevato, A., Kim, D. & Tretola, B. (2009), Comparing effective and ineffective behaviors of student programmers, *in* 'Proceedings of the fifth international workshop on Computing education research workshop', ACM, pp. 3–14.
- El-Halees, A. (2009), 'Mining students data to analyze learning behavior: A case study', *Department of Computer Science, Islamic University of Gaza PO Box* **108**.
- Falkner, N. & Falkner, K. (2012), A fast measure for identifying at-risk students in computer science, *in* 'Proceedings of the ninth annual international conference on International computing education research', ACM, pp. 55–62.
- Fenwick Jr, J., Norris, C., Barry, F., Rountree, J., Spicer, C. & Cheek, S. (2009), 'Another look at the behaviors of novice programmers', *ACM SIGCSE Bulletin* **41**(1), 296–300.
- Georg, W. (2009), 'Individual and institutional factors in the tendency to drop out of higher education: a multilevel analysis using data from the konstanz student survey', *Studies in Higher Education* **34**(6), 647–661.
- Herbert, N. & Wang, Z. (2007), Student timesheets can aid in curriculum coordination, *in* 'ACM International Conference Proceeding Series', Vol. 239, pp. 73–80.
- Lopez, M., Luna, J., Romero, C., Ventura, S., Molina, M., Luna, J., Romero, C., Ventura, S., Cano, A., Luna, J. et al. (2012), Classification via clustering for predicting final marks based on student participation in forums, *in* 'Proceedings of the 5th International Conference on Educational Data Mining, EDM 2012', Vol. 42, pp. 649–656.
- McGregor, H., Saunders, S., Fry, K. & Tayler, E. (2000), 'Designing a system for the development of communication abilities within an engineering context', *Australian Journal of Communication* **27**, 83–94.
- Merceron, A. & Yacef, K. (2005), Educational data mining: a case study, *in* 'Proceeding of the 2005 conference on Artificial Intelligence in Education: Supporting Learning through Intelligent and Socially Informed Technology', IOS Press, pp. 467–474.
- Moodle (2007), 'version 1.9'.
URL: <http://moodle.org>
- Morris, L., Finnegan, C. & Wu, S. (2005), 'Tracking student behavior, persistence, and achievement in online courses', *The Internet and Higher Education* **8**(3), 221–231.
- Murphy, C., Kaiser, G., Loveland, K. & Hasan, S. (2009), Retina: helping students and instructors based on observed programming activities, *in* 'ACM SIGCSE Bulletin', Vol. 41, ACM, pp. 178–182.

MySQL (2012), 'Mysql: the world's most popular open source database'.

URL: <http://www.mysql.com>

Nandi, D., Hamilton, M., Harland, J., Warburton, G., Hamer, J. & de Raadt, M. (2011), How active are students in online discussion forums?, in 'Proceedings of the Australasian Computing Education Conference (ACE 2011)', Australian Computer Society Sydney, pp. 125–134.

Norris, C., Barry, F., Fenwick Jr, J., Reid, K. & Rountree, J. (2008), Clockit: collecting quantitative data on how beginning software developers really work, in 'ACM SIGCSE Bulletin', Vol. 40, ACM, pp. 37–41.

Python (2012), 'Python'.

URL: <http://www.python.org/>

R Core Team (2012), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

URL: <http://www.R-project.org>

Ramsden, P. (2003), *Learning to Teach in Higher Education*, RoutledgeFalmer, London.

Sheard, J., Carbone, A., Markham, S., Hurst, A., Casey, D. & Avram, C. (2008), Performance and progression of first year ict students, in 'Proceedings of the Tenth Australasian Computing Education Conference (ACE 2008)'.

Sheard, J., Ceddia, J., Hurst, J. & Tuovinen, J. (2003), 'Inferring student learning behaviour from website interactions: A usage analysis', *Education and Information Technologies* **8**(3), 245–266.

Zhang, H., Almeroth, K., Knight, A., Bulger, M. & Mayer, R. (2007), Moodog: Tracking students on-line learning activities, in 'Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications', pp. 4415–4422.

A Comparative Analysis of Results on Programming Exams

James Harland, Daryl D'Souza and Margaret Hamilton

School of Computer Science and IT, RMIT University
 GPO Box 2476, Melbourne, 3001, Australia
 {james.harland,daryl.dsouza,margaret.hamilton}@rmit.edu.au

Abstract

Measuring student performance on assessments is increasingly important, especially when mapping outcomes to particular topics in a university subject. In this paper we investigate the classification of exam questions. In particular, we examine the performance of students in two programming subjects, as a means of determining how we can measure the difficulty of a particular question. This can not only serve as a calibration for the expectations of instructors about the difficulty levels, but also as a means of examining what it means for a question to be considered difficult.

Keywords: BABELnot, Exam classification, Assessment measurement

1 Introduction

In Australia, there is a push to measure and compare educational institutions. School data from 10,000 schools around Australia has been published in the MySchool website¹ and universities are being asked to be more accountable for their funding and to document academic standards clearly. Universities are being assessed both by our students and by our funding bodies.

The work reported in this paper contributes to the overall aims of the BABELnot project (Lister et al. 2012), which commenced in 2011. It is funded via a grant from the Office of Learning and Teaching (OLT), and, in a nutshell, aims to develop a common language in which educators of programming in ICT degrees may better communicate about assessments and standards within their subjects.

Programming has long been regarded as a learning bottleneck for novices, typically students entering their first semester of their ICT degrees. High failure and attrition rates are commonplace, and a lot of energy has been spent on research to understand the reasons but yet often the causes of such outcomes have been explained away on the basis of opinion and folklore (Sheard et al. 2009). The BABELnot project seeks to develop an epistemology or common understanding of programming concepts and ways of assessing competency across programming subjects.

In the School of Computer Science & IT at RMIT University, we have several programming subjects,

varying from introductory ones to advanced ones (requiring two or three previous semesters of programming experience). All the subjects involve the learning of programming, but use Python, Java, PHP and C as the primary coding languages, and offer different outcomes for undergraduate or postgraduate students, as well as for various other students taking a programming stream. For instance, a student may study multimedia, or games, or engineering, all of which have degree programs which contain a programming stream consisting of three programming subjects. Also, a student may be undertaking an undergraduate program of three years, or a postgraduate program of one year and still require a knowledge of programming for their ICT qualification. Hence the outcomes required by students when undertaking the programming stream differ immensely.

The context of each subject makes it very unclear where to ask to insert various exam questions. Another factor is the readiness of the subject manager to allow exam questions from a different origin. The learning outcomes of the various subjects are sufficiently different to introduce confusion about whether or not a particular question will be testing that outcome. Finally the mixture of concepts required to answer the question makes it difficult to be able to place it within the context of any particular subject.

However, despite all of the above difficulties, we approached all the subject managers and asked if any would be interested in putting any of our BABELnot exam questions into their exam papers for that semester. Two lecturers agreed, and their answers provide the results which we discuss here in this paper.

Several issues arise. The stated outcomes and capabilities in subject guides are typically vague, a problem being addressed elsewhere in the BABELnot project, and often lead to dichotomous teacher-student perceptions about the questions in exams. Teachers will of course believe that they are using the summative assessment instrument appropriately, and more often than not their attempts are genuine and sincere in attempting to meet the broadly expressed outcomes.

On the other hand, and particularly at the introductory programming stage, students do not necessarily handle summative assessments as their lecturers expect them to (Shuhidan et al. 2010, Tew and Guzdial 2010). Typically high failure rates demonstrate that a significant number of students find exams hard (Tew and Guzdial 2011).

In this paper we investigate how we may use student performance on exam questions as an indicator of the difficulty of the question. In particular, we wish to be able to relate the *Degree of difficulty* measure used in the BABELnot classifications (Sheard

Copyright ©2013, Australian Computer Society, Inc. This paper appeared at the 15th Australasian Computer Education Conference (ACE 2013), Adelaide, South Australia, January-February 2013. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 136, Angela Carbone and Jacqueline Whalley, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

¹www.myschool.edu.au

et al. 2011) to student performance, and hence improve the accuracy of our classification. There are a number of measures used to classify exam questions, many of which come with some associated criteria for their application (Simon et al. 2012). The Degree of difficulty is intended as a holistic measure, and hence does not come with any specific criteria. This is also considered the measure most likely to correlate with student performance. Hence our focus is on how we may measure the difficulty of a question by the students' performances on it, and how this relates to the BABELnot classifications.

Another aim is for a standard approach to the reporting of exam performance, and in particular the statistical information that will best provide for the classification of questions. There are many issues to be considered in the development of such a standard (such as those discussed by Anfoff (Angoff 1971)), and the approach reported in this paper is not necessarily going to become a standard one, but it should be seen as a starting point for discussion.

This paper is organised as follows. In Section 2 we discuss the background to our work, and the BABELnot project. In Section 3, we describe the two subjects used in our research, and our approach to analysing the exam results. In Section 4 we present our data, and in Section 5 we discuss the implications of it. Finally in Section 6 we present our conclusions and some areas of further work.

2 Background

As indicated earlier this work contributes to the BABELnot project, which was funded in 2011 by the Office of Learning and Teaching (OLT) (<http://www.olt.gov.au>). The BABELnot project held its inaugural major meeting in October of 2011 in Melbourne and is funded through to August 2013.

The project was prompted by a need to document academic standards associated with a sequence of up to three programming subjects in six participating universities (UTS, QUT, Monash, RMIT and the universities of Sydney and Newcastle) and optionally other universities as well. In order to meet this objective, two broad subgoals were presented. One was to “develop a rich framework for describing the learning goals associated with programming” and the other was to “benchmark exam questions that are mapped onto this framework”. Further details about the rationale for the project are documented in Lister et al (Lister et al. 2012). The project unified several projects in existence at the time of commencement: Exam Question Classification, Syllabus Specification, Exam Question Generation and Benchmarking, and Neo-Piagetian Theory (and its application to the learning of programming).

This paper contributes to the Exam Question Generation and Benchmarking component of the BABELnot project. A significant precursor to this component of BABELnot was the BRACElet project (Clear et al. 2010). The aim of BRACElet was to collect evidence from end-of-study-period exams to determine what were the novice programmers' problems that caused high failure rates and high attrition rates. This aim served to establish scientific evidence about the difficulties faced by novice programmers, in the face of prevailing folklore “evidence”. The BRACElet project concluded amongst other things that students tend not to have problems with low level aspects of programming, but with the larger picture of piecing together the parts into a whole, or of not being able to “see the forest for the trees” (Lister et al. 2012).

The BRACElet project led to a formal investigation of exam classification in the Exam Question Classification and Benchmarking project. To that end, little further work has been done in the area of benchmarking (Shuhidan et al. 2010).

The ratings used in the BABELnot project involve a number of different measures. These include

Topics covered
Skill required
Style of question
Open or closed
Degree of difficulty
External domain references
Explicitness
Linguistic complexity
Conceptual complexity
Intellectual complexity (Bloom level)
Code length

The measures for *Topics covered* and *Skills required* come with a list of pre-defined topics and skills from which a small number must be chosen by the classifier. The measure for *Conceptual complexity* comes with a mapping between particular concepts covered, and the three ratings of *low*, *medium* and *high*. The *Intellectual complexity* measure is based on Bloom's taxonomy. The measure for *Degree of difficulty* is intended as a holistic measure, and one that should correlate the most with student performance (although this measure could and should correlate to some degree with other measures, such as *Intellectual complexity* and *Conceptual complexity*).

3 Exam Questions

In this section we describe our approach to the analysis of exam questions, and some details of the two chosen subjects.

3.1 Methodology

There are a number of ways in which exam results could be analysed (Angoff 1971, de Klerk 2008). Our intention is to inform our classification of exam questions by student performance results, and in particular to see how our perceptions of difficulty align with the marks obtained by the students on particular questions or groups of questions.

In order to do this, we first look at the overall grade distribution. This gives us some insight into the overall difficulty of the exam, as well as the distribution of ‘student types’, i.e. how many high-achieving students there are compared to those who barely pass. The most obvious categorisation of student types is by the grades they achieve, and so we will classify students according to their grades, rather than a finer-grained scheme (e.g. the percentage decile in which their mark falls) or a coarser-grained one (e.g. whether they passed the exam or not).

The relevant grades are given in the table below.

Grade	Mark	Name
HD	$\geq 80\%$	High Distinction
DI	$< 80\%$ and $\geq 70\%$	Distinction
CR	$< 70\%$ and $\geq 60\%$	Credit
PA	$< 60\%$ and $\geq 50\%$	Pass
NN	$< 50\%$ and $\geq 25\%$	Fail
FF	$< 25\%$	

FF is not actually a separate grade from NN. However, we separate the data here for analysis purposes (and specifically to try to isolate ‘genuine’ failures

from those who made no serious attempt). As it turns out, there were comparatively few of these students (4 out of 236 in P1, and 6 out of 160 in PT), and hence their effect on our results is minimal.

We then investigate the students' performance on each question or question type. Firstly, we look at the mean and median marks for each question (and the mode, if applicable). This gives us some indication of the performance of the students on each question, as well as the range of marks. For example, there are some questions in which the mean mark is well below the median, which indicates that there is a wide spread of marks (as if at least 50% of the students scored better than the average, then the lower scoring students must have scored generally very low marks). In cases where the median mark is 100%, this indicates that at least half the students got full marks.

Our next two analyses are similar in spirit, but differ on some detail. One analysis looks at performance on each question by the different classes of students, as classified by their grades. Hence we look at how the HD students performed on each question or question type, as measured by the average mark obtained on the question by the HD students. We then perform the same analysis for the DI students, the CR students and so on. This effectively gives us a profile of the performance of a 'typical' HD, DI, CR, PA, NN or FF student, and in particular how this question can be rated by the students' performance on it (e.g. "The best students generally struggled with this question, while the weaker students found it very difficult").

The other analysis is to consider the number and range of marks obtained for each question. We do this by looking at the number of students who scored 80 or more on this question, and we label these students as HD, those who obtained between 70% and 80% as DI, and so forth, giving us an idea of the distribution of the performance of the overall student population on each question. This will help inform us about the rating of questions by allowing us to draw conclusions such as "Very few students got full marks for this question" or that "Most students got at least 50% on this question".

Hence the difference between these two latter analyses is that the former divides the student population into classes, and then looks at the performance of each class on specific questions, whereas the latter one looks at the spread of student marks for each question.

3.2 Subjects

Both of the subjects whose results are presented here were taught in semester 1, 2012 at RMIT University.

Programming 1 is a first programming subject in Java, with no previous knowledge of programming necessary. There were 236 students in semester 1, 2012 who sat for the exam in this subject. In BABELnot terms, this is a level 1 subject.

The Programming 1 exam consisted of three main parts:

1. 20 multiple choice questions (30 marks)
2. 6 short answer questions (35 marks)
3. 3 interrelated programming problems (35 marks)

We will refer to each of these three parts as **MCQ**, **Short** and **Classes** respectively.

Programming Techniques assumes some significant programming experience, and specifically two

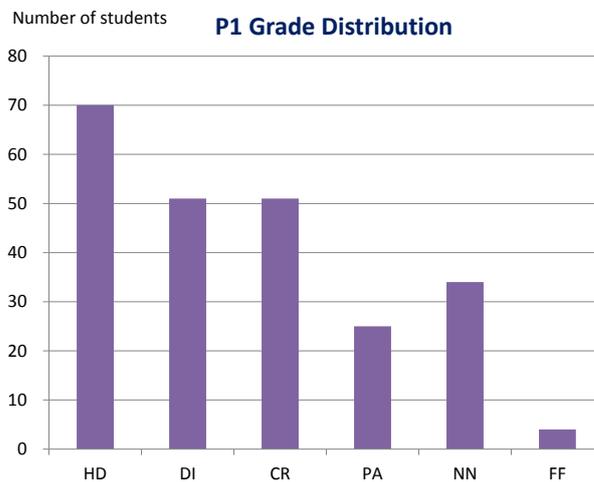


Figure 1: Programming 1: Overall grades

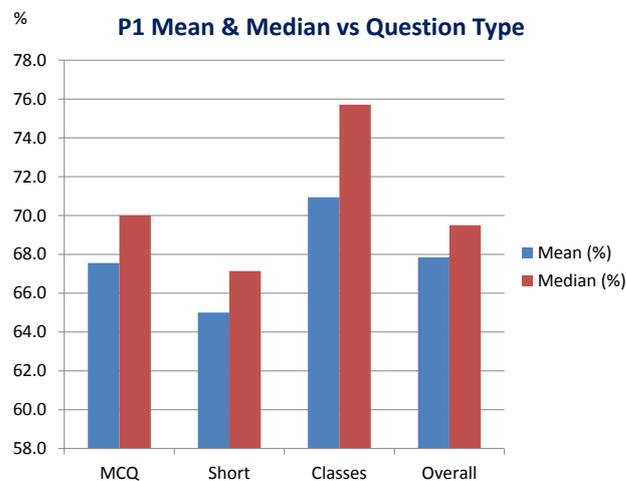


Figure 2: Programming 1: Mean & Median

previous semesters of programming in Java. This subject is taught in C, and 160 students sat for the exam in semester 1, 2012. In BABELnot terms, this is a level 3 subject.

The Programming Techniques exam consisted of

1. 7 short answer questions (35 marks)
2. 9 programming problems (145 marks)

4 Exam Results

In this section we discuss the results for Programming 1 and Programming Techniques.

4.1 Programming 1

The results for Programming 1 are presented in Figures 1, 2, 3 and 4.

The graph in Figure 1 ('P1 Grade Distribution') shows the overall distribution of grades. This conforms to a typical pattern for introductory programming subjects, i.e. showing some bipolar tendencies. 70 students out of the total of 236 (or almost 30%) got a grade of HD, whilst around 16% failed. Of the passing grades, PA had the lowest proportion of students at 10%.

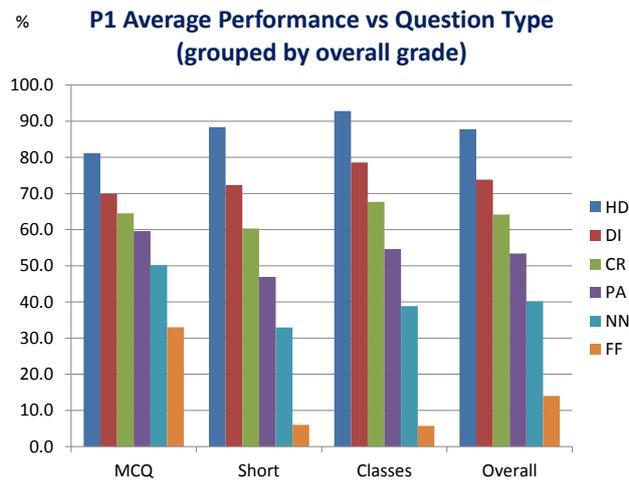


Figure 3: Programming 1: Student type performance

The graph in Figure 2 ('P1 Mean & Median') shows the mean and median mark on each of the three sets of questions (the mode mark for all three was zero). One point to note is that the average mark on all three question sets was above 60%. Given that the median is higher than the mean on all three question sets, this is some evidence of a wide spread of marks, as if at least 50% of the students scored better than the average, then the lower scoring students must have scored generally very low marks.

It is perhaps a little surprising that the highest average mark is for the **Classes** questions, which are the ones that would be expected to be the most difficult. This is probably explained by the fact that the Programming 1 students were informed in advance of the scenario on which the questions would be based, and the students were given code skeletons which they had to complete. This was a new approach compared to previous years, in which students were not informed in this way, and they had to write complete pieces of code.

The graph in Figure 3 ('P1 Student type performance') shows the performance of the students, classified by their grade, against the three question types and their overall results. In other words, the students who obtained an HD grade scored an average of just over 80% on the **MCQ** questions, just under 90% on the **Short** questions and just over 90% on the **Classes** questions, whilst averaging 88% overall. Those who obtained a DI grade had a similar pattern of also performing best on the **Classes** questions. The CR and PA students were at their worst on the **Short** questions, unlike the HD and DI students, for whom the **MCQ** questions were their worst performance. This is a little counter-intuitive, in that one may expect the better students to perform best on all questions; further analysis of this data and comparison with similar exams are some items of future work. Less surprisingly, the NN and FF students performed best on the **MCQ** questions.

An interesting property of the **Classes** questions is that they appear to have an "Eden-Monaro" property, i.e. that performance on this question mirrors accurately the students' performance on the exam overall.² This property is that the students who obtained HD overall scored at least 80% on this question, those

²Eden-Monaro is an Australian Electoral Division which has been held by the government of the day since 1972, and has gained a reputation as an indicator of national voting trends.

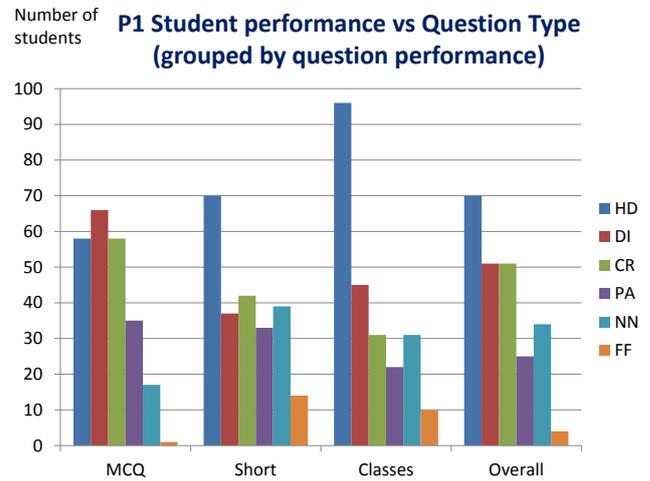


Figure 4: Programming 1: Student performance

who obtained DI scored between 70% and 79%, those who obtained CR scored between 60% and 69%, those who obtained PA scored between 50% and 59% and those who failed the exam scored less than 50%. This is perhaps not altogether unsurprising given that the **Classes** questions are worth 35 marks out of a total of 100, but it is also worth noting that neither of the other two question sets, worth a total of 65 marks overall, had this property.

The graph in Figure 4 ('P1 Student performance') shows the performance of the students on each question. Hence nearly 100 students got 80% or more on the **Classes** questions, with just under 60 students getting 80% or more on the **MCQ** questions. This shows perhaps most starkly that the students performed best on the **Classes** questions.

In order to compare the effectiveness of the three question sets (**MCQ**, **Short**, **Classes**) as means of classifying students, we performed a χ^2 test comparing performance on the individual question sets compared with the students overall performance. This returned *Pr* values of 0.027, 0.072 and 0.032 for the sets **MCQ**, **Short** and **Classes** respectively. This is in some ways a biased test, in that each class component forms a part of the overall performance, but interestingly only the **Short** class returned a *Pr* value over the traditional threshold of 0.05 for statistical significance. From this we conclude that the **Short** class of questions was a more accurate prediction of the students' overall performance than either of the other two question sets.

We also performed similar tests comparing each pair of question sets. The only one of these three pairs to return a non-zero *Pr* value was for **Short** and **Classes**, for which the value was 0.067. This suggests that there is some correlation between performance on these sets of questions, but that the performance on **MCQ** does not reflect performance on the other sets.

4.2 Programming Techniques

The results for Programming Techniques are presented in Figures 5, 6, 7, 8 and 9.

The graph in Figure 5 ('PT Grade Distribution') shows the overall distribution of grades. This conforms to a typical pattern for advanced programming subjects, with a large number of students getting a grade of HD. This is at least partly due to students

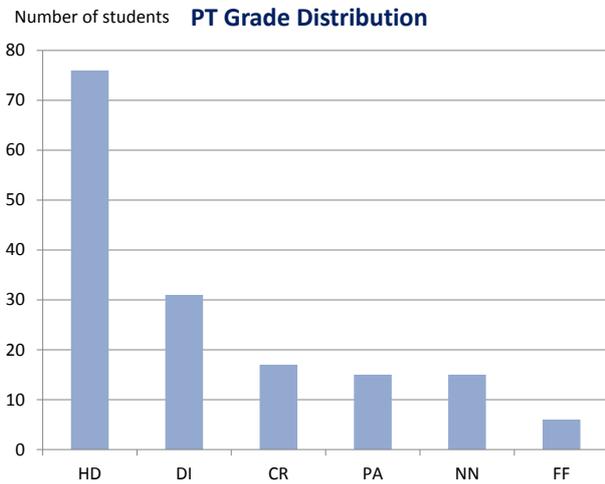


Figure 5: PT: Grade Distribution

having to have completed two prior semesters of programming before entering this subject, and so a relatively advanced level of programming ability is a prerequisite.

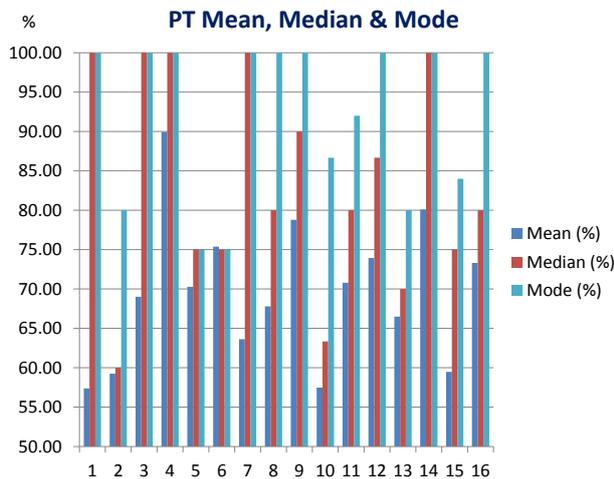


Figure 6: PT: Mean, Median & Mode

The graph in Figure 6 ('PT Mean, Median & Mode') shows the mean, median and mode for each question. Note the scale of the graph, in which the lowest value shown is 50%, reflecting that the mean, median and mode for each question was at least 50% (and in fact the lowest mean value was 57%). From this it seems reasonable to conclude that Question 4 was the one the students found easiest, as it had the highest mean (of 90%) and with the median and mode both being 100%. Questions 1, 3, 4, 7 and 14, being those for which the median and mode are 100% could also be argued to be ones on which performance showed some variability, with Question 1 (the one with the lowest mean) the most varied of all. Questions 5 and 6, in contrast, had the most uniform performance, with the mean, median and mode marks all virtually identical. It is also notable that Questions 2 and 10 are arguably the most difficult, with the two lowest median marks. The high modes for each of these questions suggests that the distribution is bi-polar rather than spread, i.e. that most student either got most of the marks, or very few

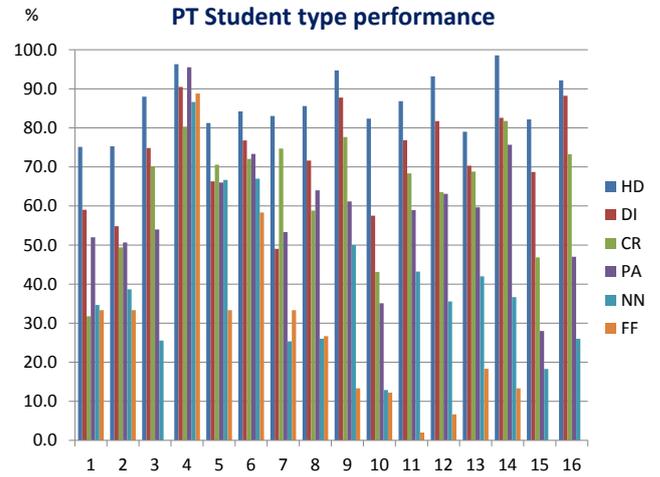


Figure 7: PT: Student type performance

marks.

The graph in Figure 7 ('PT Student type performance') shows the performance of the students, classified by their grade, against all questions. The HD students, as expected, were the best performed students on all questions. However, the questions on which the average marks were lowest were Questions 1 and 2. It is clear from this graph that Question 4 can be considered the easiest question, as every class of student scored 80% or more for this question. The fact that the CR students performed slightly worse than the PA, NN and FF students is presumably due to noise, rather than indicative of anything more significant. It is also noticeable that the only questions on which NN or FF students scored an average of 50% or more were Questions 4, 5, 6 and 9, which also suggests that these were easier questions than the others. Question 14 is also arguably an easier question, given that the HD students got an average mark of 99% on it, and it was also the question on which the PA students obtained their second-highest average mark (after Question 4). However, the NN and FF students didn't score particularly well on this question. As the PA students had an average mark of 75%, this question seems to have a 'polarising' quality, in that students who passed the exam generally did well on it, but those who failed tended to score poorly on it. Similar comments apply to Questions 3, 8, 11, 12 and 13. Question 7 is almost in the same class, except for the rather puzzling 'reversal' of performance of the DI and CR students on this question (the DI students averaged 49%, the CR students 75%).

Question 15 is arguably a moderately challenging question, in that the only classes of students with an average mark of 50% or more were the HD and DI students. This has a similar polarising effect, but this time in separating the better students from those who barely pass or fail. Question 10 is similar. Questions 1 and 2 are similar, although the division between the HD students and the DI students is starker, and the performance of the CR and PA students is more variable on Questions 1 and 2, making them seemingly less reliable polarisers.

Question 16 could also be considered a polarising question, in that the HD, DI and CR students all performed relatively well, but with the PA students scoring an average of 47%.

Question 11 has the Eden-Monaro property mentioned above (i.e. a student's performance on this

question accurately reflects his or her overall mark). Questions 3 is similar, although not quite perfect given that the CR students have an average mark of 70%.

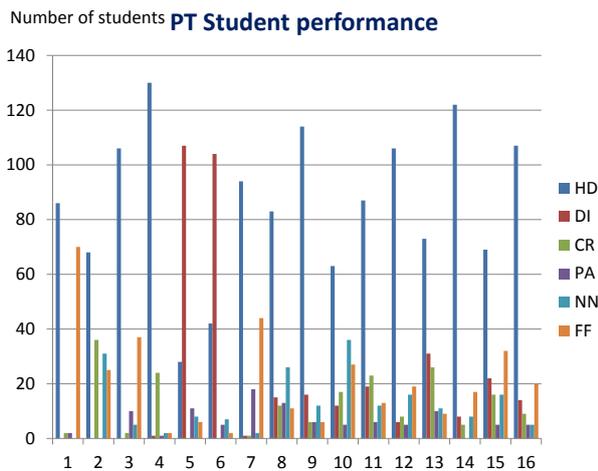


Figure 8: PT: Student performance

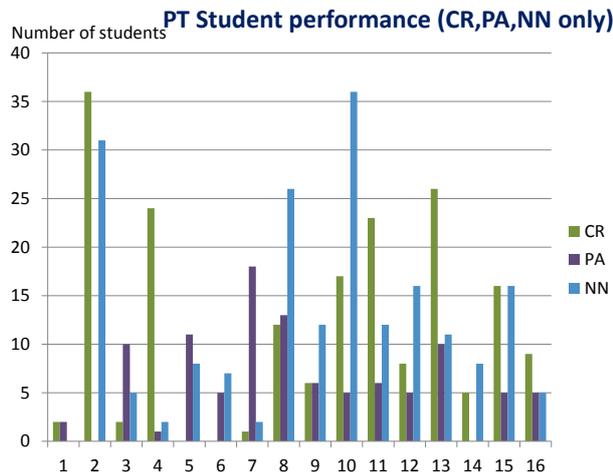


Figure 9: PT: Student performance (CR, PA, & NN only)

The graph in Figure 6 ('PT Student performance') show the performance of the students on each question. The graph in Figure 9 shows only the performance levels of CR, PA and NN, as these are obscured somewhat in the upper figure by the relatively high values for HD, DI and FF. It is easily seen that the proportion of students getting HD on each question is very high, with the exceptions of Questions 5 and 6, suggesting that these were a little more difficult (or more likely that there was a minor loss of marks that was very common). One could also imagine a metric for difficulty based on the number of HD scores on the question, which would indicate that the 5 easiest questions (in descending order) were Questions 4, 14, 9, 16 and 3, although if the metric was the number of HDs and DIs on a particular question, then Questions 5 and 6 would be the two easiest. It should also be noted that as Questions 1 to 7 were marked out of 5, there is a more limited scope for distinguishing between passing grades (ie between HD, DI, CR and PA) than on other questions.

Question 1 clearly had a polarising effect, in that almost all students got either over 80% or under 25%. Questions 3 and 7 are similar, without being quite as extreme, and if one considers the HD and DI population together, Questions 5 and 6 also had this effect. One could also argue that Questions 1, 7, 3, 15 and 2 were the most difficult, on the grounds that these were the questions with the greatest number of students who scored in the FF range

We also performed χ^2 tests for each of Questions 1 to 16 compared to the overall result. The only values that were not 0 or 0.001 were those for Questions 8, 11 and 13 with values of 0.046, 0.046 and 0.527 respectively, of which only the value of 0.527 for Question 13 is above the threshold value of 0.05 for significance. Hence student performance on Question 13 seems to be a very good predictor of overall performance, with Questions 8 and 11 not at the same level, but perhaps having some indication of ability, especially when compared to all of the other questions.

5 Discussion

As mentioned above, we are interested in investigating the *Degree of difficulty* measure. This is intended as a holistic measure, and one that should correlate the most with student performance (although this measure could and should correlate to some degree with other measures, such as *Intellectual complexity* and *Conceptual complexity*). We may see the results discussed above as a measure of the *Degree of difficulty* more than anything else. Given that the rating is one of the three values *low*, *medium* and *high*, it would seem reasonable to use the above data to determine a way of classifying questions on this scale according to student performance (such as an average mark of 80% indicating that the question is of low difficulty).

Turning to the Programming 1 result discussed above, it seems that measuring *Degree of difficulty* by looking at the average mark on a question is too simplistic; it is important to look at the spread of results and whether the question is 'polarising' students or not. In relative terms, **Classes** was the one the students found easiest, with **MCQ** next and then **Short**. The better students (HD, DI) tended to score better marks on **Short** than on **MCQ** (a situation that is reversed for every other type of student), but it is arguable that this is a property of more difficult questions. Certainly a question on which the top students struggle to get more than 50% would be considered difficult; is a question difficult if the top students do well, but other students struggle? In the case of **Short**, it would seem that this would be considered *medium*, as it doesn't seem to be *low*, and to class it as *high* when nearly half the students score more than 70% doesn't seem right. This seems to leave the only option for **MCQ** to be also *medium*, as it is hard to argue that it is more difficult than **Short**, but calling it *low* seems to contradict that this was the question on which the HD and DI students scored least well. This would make **Classes** either *low* or *medium*, depending on how significant one feels the Eden-Monaro property is (with greater significance indicating a stronger likelihood that it is *medium*).

Turning to Programming Techniques, it seems clear that Questions 4 and 6 should be classified as low (if for no other reason that the FF students scored nearly 90% on Question 4 on average, and over 50% on Question 6). One could also argue that Question 5 should be rated as low, as on this question, the

NN students scored more than 50% on average (and Questions 4, 5, 6 and 9 were the only ones with this property). On the strong overall performance of the students, it would seem a long bow to draw to classify any of the question as *high*. The hardest questions, according to the mean mark, were Questions 1, 10, 2 and 15 respectively, and on these questions the HD and DI students generally did well, whilst the others struggled. This is analogous to the **Short** question on Programming 1. Note also that there is a polarising effect here, strongest in Question 1, but present to some degree in Questions 10 and 15. Hence it is tempting to classify Questions 4,5,6 and 9 as the easiest group, Questions 1, 2, 10 and 15 as the hardest, and the remaining 8 questions somewhere in the middle. It seems hard to argue that Questions 1, 2, 10 and 15 are of *high* difficulty, and at least as hard to argue that they are *low*. It seems reasonable to argue that Questions 4, 5 and 6 (and possibly 9) are *low*, but this would make all others (in this case at least 12 out of 16 questions) *medium*.

Questions 7 and 12 on the Programming Techniques exam were drawn from the BABELnot repository, as were Questions 5, 11 and 12 from the MCQ questions on the Programming 1 exam. In fact, Question 12 on the Programming Techniques exam and Question 11 on the MCQ questions were basically the same, only that the Programming 1 version was scaffolded to a much greater degree, and the students only had to select which lines of code had to be changed. The Programming Techniques students had to write complete (C) code for the specified function.

We have not presented the data for the individual parts of the MCQ questions, and so it is hard to make a direct comparison between performance on the **Programming 1** version of the question compared to the one for **Programming Techniques**. However, we note that Question 12 on the **Programming Techniques** exam had a mode of 100%, an average mark of 74% and a median of 86%. Moreover, the passing students (i.e. the HD, DI, CR, and PA students) all did well on this question, whilst the NN and FF students scored an average of 42% and 7% respectively. Hence, whilst it may not be the easiest question on the paper, it was in the easier half, at least. It is also worth noting that 49% of the **Programming 1** students got the correct answer for MCQ Question 11. This lower level of performance together with the simpler nature of the question being asked tends to indicate that the **Programming 1** students found this question considerably more difficult than their **Programming Techniques** counterparts, as would be expected.

6 Conclusions and Further Work

We have discussed the exam results of two programming subjects, one of which assumes no programming background, and the other requiring two semesters of programming experience. We have seen how attempting to align student performance on these exams with the BABELnot notions of *Degree of difficulty* has led to some complications. On the basis of these results, it seems reasonable to conclude that an absolute scale of *low*, *medium* and *high* is not appropriate for classifying questions (or at least not for classifying questions based on students' results). The underlying issue is that this measure is attempting to summarise student performance in an overly simplistic way; it seems that the distribution of students' marks contains some features which are not readily apparent in a three-point scale.

Perhaps the most intuitive way to characterise the difficulty of questions is to look at student types, and to determine the difficulty of the question relative to the performance of each student type. For example, if the weakest students (as measured by overall performance on the exam) get an average of 90% or more, it seems inescapable that the question is of low difficulty. However, when the performance of each student type diverges, it can be more problematic. For example, if half of the student get full marks and the other half get 0, how difficult is the question? It would seem that it is better to classify such a question as a 'perfect polariser' rather than being of a particular level of difficulty.

It certainly seems that the only way to make progress on issues like these is to continue analyses of this sort on an increasingly large set of data. This would allow more detailed and specific criteria to emerge, and possibly also a more refined understanding of the notion of difficulty.

One item of future work is to perform a further analysis of the data from these two exams, such as measuring the internal consistency of the questions in each exam, as suggested by de Klerk (de Klerk 2008), using inter-correlations as measured by Cronbach's Alpha. Another issue discussed by de Klerk is the validity of testing, which in our case corresponds to determining the appropriate level of content for level 1 and level 3 programming exams. Another possibility is to determine the difficulty of a given question in comparison to a normative sample, or standard group for comparison purposes. Both of these aspects will be informed and enhanced by the work of the BABELnot project.

References

- Angoff, W., (1971), *Scales, norms, and equivalent scores*, in R.L. Thomdike (Ed.), *Educational measurement* (2nd ed., pp. 508-600). Washington, DC: American Council on Education.
- Clear, T., Whalley, J., Robbins, A., Philpott, A., Eckerdal, A., Laakso, M.-J., and Lister, R., (2011), *Report on the final BRACElet workshop*, Auckland University of Technology, September 2010, *Journal of Applied Computing and Information* 15(1).
- de Klerk, G., *Classical test theory (CTT)*, in M. Born, C.D. Foxcroft & R. Butter (Eds.), *Online Readings in Testing and Assessment*, International Test Commission, <http://www.intest.com.org/Publications/ORTA.php>.
- Lister, R., Clear, T., Simon, Bouvier, D., Carter, P., Eckerdal, A., Jackova, J., Lopez, M., McCartney, R., Robbins, A., Seppala O., and Thompson, E., (2011), *Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer*, *SIGCSE Bulletin* 41:156-173.
- Lister, R., Corney, M., Curran, J., D'Souza, D., Fidge, C., Gluga, R., Hamilton, M., Harland, J., Hogan, J., Kay, J., Murphy, T., Roggenkamp, M., Sheard, J., Simon and Teague, D., (2012), *Towards a shared understanding of competency in programming: An invitation to the BABELnot project*, in Proceedings of the Fourteenth Australasian Computing Education Conference (ACE2012) 53-60, Melbourne, Australia.
- Lister, R., (2011), *Concrete and other neo-Piatetian forms of reasoning in the novice programmer*, *Thir-*

teenth Australasian Computing Education Conference (ACE2011), Perth, Australia, 9-18.

Petersen, A., Craig, M., and Zingaro, D., (2011), *Reviewing CS1 exam question content*, SIGCSE 2011, Dallas, Texas, USA, 631-636.

Schulte C., and Bennedsen, J., (2006), *What do teachers teach in introductory programming?* Second International Computing Education Research Workshop (ICER 2006), Canterbury, UK, 17-28.

Sheard, J., Simon, Hamilton, M., and Lonnberg, J., (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, Carbone, A., Chinn, D., Laakso, M.-J., Clear, T., de Raadt, M., D'Souza, D., Harland, J., Lister, R., Philpott, A., and Warburton, G., (2011), *Exploring programming assessment instruments: a classification scheme for examination questions*, Seventh International Computing Education Research Workshop (ICER2011), Providence, RI, USA, 33-38.

Shuhidan, S., Hamilton, M., and D'Souza, D., (2010), *Instructor perspectives of multiple choice questions in summative assessment for novice programmers*, Computer Science Education 20:229-259, 2010.

Simon, Sheard, J., Carbone, A., Chinn, D., Laakso, M.-J., Clear, T., de Raadt, M., D'Souza, D., Lister, R., Philpott, A., Skene, J., and Warburton, G., (2012), *Introductory programming: examining the exams*, Fourteenth Computing Education Conference (ACE2012), Melbourne, Australia.

Elliott Tew, A., and Guzdial, M., (2010), *Developing a validated assessment of fundamental CS1 concepts*, Proceedings of 41st SIGCSE Technical Symposium on Computer Science Education 97-101, Milwaukee, 2010.

Elliott Tew, A., and Guzdial, M., (2011), *The FCS1: A language independent assessment CS1 concepts*, Proceedings of the 42nd SIGCSE Technical Symposium on Computer Science Education 111-116, Dallas, 2011.

Appendix: Programming Techniques Exam Questions

QUESTION 7

In one sentence, explain the purpose of the following piece of code.

```
int fn(int *array, size_t n)
{
    size_t i;

    for (i = 0; i < n - 1; i++) {
        if (array[i] > array[i + 1])
            return -1;
    }

    return 1;
}
```

QUESTION 12

The purpose of the block of code below is to take an array of integers and move all elements of the array one place to the right, with the rightmost element moving around to the leftmost position.

```
void shift_right(int *array, int n)
{
    int temp = array[n - 1];

    int i;

    for (i = n - 2; i >= 0; i--)
        array[i + 1] = array[i];

    array[0] = temp;
}
```

Write a function `shift_left` that will move all elements of the array one place to the left, with the leftmost element moving around to the rightmost position.

```
void shift_left(int *array, int n)
{
    /* ... */
}
```

Appendix: Programming 1 Exam Questions

QUESTION 5 (1.5 Marks) Consider the following method:

```
boolean testArray(int [] x, int arrayLength)
{
    for (int i = 0; i < arrayLength-1; i++)
    {
        if (x[i] > x[i+1]) return false;
    }
    return true;
}
```

If testArray returns true after this code is executed, which of the following is the strongest statement we can make about the contents of the array x? Assume the subscripts p, p+1, and q are legal indexes of x.

- A) $x[p] \leq x[q]$ for all $p < q$.
- B) $x[p] \leq x[p+1]$
- C) $x[p] \leq x[p+1]$ where p is an even number (i.e. 0, 2, 4, etc).
- D) No statement can be made, as the last iteration of the loop will attempt to index an element of the array that does not exist.
- E) $x[0] < x[arrayLength-1]$

QUESTION 11 (1.5 Marks)

The code below in the right of the table moves all elements of the array x one place to the **right**, with the **rightmost** element being moved to the **leftmost** position. The variable length contains the number of elements in the array x:

Line	left	right
1	int temp = ???	int temp = x[length-1];
2	for (int i ???	for (int i=length-2; i>=0; i--)
3	x[i-1] = x[i];	x[i+1] = x[i];
4	??? = temp;	x[0] = temp;

Consider the partial code provided in the above table, on the left. When the occurrences of ??? are replaced with appropriate code, that code can undo the effect of the code on the right. That is, when the ??? are replaced appropriately, the code can move all elements of the array x one place to the **left**, with the **leftmost** element being moved to the **rightmost** position.

The table already shows that line 3 is different in the code on left to the code on the right. When the ??? on lines 1, 2 and 4 are replaced with appropriate code, which of these lines of code **must** be different between the left and the right

- A) None of the lines 1, 2 and 4 **must** be different.
- B) Only lines 1 and 4 **must** be different.
- C) Only line 2 **must** be different.
- D) All of the lines 1, 2 and 4 **must** be different.

QUESTION 12**(1.5 Marks)**

Below is incomplete code for a function which returns the minimum value in the array *x*. When an appropriate line of code is selected from each box, the completed code will scan across the array, using the variable *minsofar* to remember the best candidate for minimum so far.

```
int min(int x[], int arrayLength)
{
    int minsofar = 

|          |
|----------|
| (A) 0    |
| (B) x[0] |

 ;

    for (int i = 1 ; i < arrayLength; i++)
    {
        if ( x[i] < 

|                 |
|-----------------|
| (C) minsofar    |
| (D) x[minsofar] |

 )
        {

            minsofar = 

|          |
|----------|
| (E) i    |
| (F) x[i] |

 ;
        }
    }
    return 

|                 |
|-----------------|
| (G) minsofar    |
| (H) x[minsofar] |

 ;
}
```

Which of the following choices of lines from the boxes will produce a correct version of the function *min*?

- A) Lines ACFG only.
- B) Lines ADEH only.
- C) Lines BCFG only.
- D) Both lines ADEH and lines BCFG.

Examining Student Reflections from a Constructively Aligned Introductory Programming Unit

Andrew Cain

Clinton J Woodward

Faculty of Information and Communication Technologies
 Swinburne University of Technology,
 John Street, Hawthorn, Victoria 3122,
 Email: acain@swin.edu.au cwoodward@swin.edu.au

Abstract

Constructive alignment has been widely accepted as a strong pedagogical approach that promotes deep learning, however its application to programming units in higher education has not been widely reported. A constructively aligned introductory programming unit with portfolio assessment provides an opportunity for students to reflect on their learning. These reflections provide a rich source of information for educators looking to identify topical and pedagogical issues influencing student outcomes. In this work we applied thematic analysis to the reflective reports presented by students as part of their portfolio submission for an introductory programming unit. The analysis indicates several interesting aspects related to both topical and pedagogical issues. These results can be used to inform the development of constructively aligned programming units, and inform future research.

Keywords: Constructive Alignment, Portfolio Assessment, Reflection, Introductory Programming, Programming Issues, Thematic Analysis.

1 Introduction

Biggs' model of Constructive Alignment (Biggs 1996), based upon constructive learning theory (constructivism) and aligned curriculum (Cohen 1987), aims to enhance student learning outcomes by focusing on *what the student does*. The challenges of teaching introductory programming are widely reported (Pears et al. 2007), and it has been argued that computer science and software engineering units¹ need to transition to constructive alignment in order to better meet the requirements of the profession and improve student engagement with the field (Armarego 2009).

Recent work by Thota & Whitfield (2010) and our work in Cain & Woodward (2012) have demonstrated different methods for applying the principles of constructive alignment to the teaching of introductory programming. Thota & Whitfield (2010) presented a holistic and constructively aligned approach, aligning traditional forms of assessment and activities with cognitive and affective learning outcomes. We suggest that, in some respects, the use of traditional assessment methods can limit student engagement and constructive alignment. In Cain & Woodward (2012)

we proposed an alternative approach using portfolio assessment to create a constructively aligned introductory programming unit. It is a similar approach to that originally proposed by Biggs & Tang (1997).

A key aspect of portfolio assessment is to encourage students to reflect on their learning (Biggs & Tang 1997). In the approach we suggested in (Cain & Woodward 2012), these reflections are captured in a reflective report that is required as part of each student's portfolio. In these reflections students are required to discuss the pieces they have included in their portfolio, and how they relate to intended learning outcomes. Students are also encouraged to reflect on their learning in general.

By considering issues students are facing, as educators we are able to adapt teaching methods with the aim of improving student outcomes. In a survey by Pears et al. (2007), existing work on the teaching of introductory programming was considered and related to defined categories of curricula, pedagogy, language choice, and tools for teaching. Issues for students can be related to each of these areas. Although the last two categories are important, they are typically selected by teaching staff with little student input. The outcomes of research related to curricula and pedagogy are intrinsically more general in nature, and of use in a broader context.

A number of other studies have examined the nature of issues faced by novice programmers (Robins et al. 2003, Lahtinen et al. 2005). Some of the problems identified by Robins et al. (2003) included issues related to program design, algorithmic complexity, certain language features, and "fragile" novice knowledge. Lahtinen et al. (2005) conducted a survey of over 500 students, across a number of courses and institutions, asking them to rate various issues. Results indicated that issues with pointers and references, error handling, and recursion were comparatively ranked as more difficult than issues with selection structures, loops, and variables. Our work will build on these findings by examining issues identified by students undertaking a portfolio assessed introductory programming unit.

As part of an ongoing initiative to improve constructively aligned portfolio assessment for introductory programming, we wish to reflect on the learning outcomes presented in students' portfolios.² Student reflections provide an open opportunity to identify issues that are relevant from the students' perspective. The investigation presented in this paper analyses issues identified in student reflections from a constructively aligned, portfolio assessed, introductory programming unit, and we provide some recommendations to help inform the development of units using this approach.

This paper first outlines the method used in con-

Copyright ©2013, Australian Computer Society, Inc. This paper appeared at the 15th Australasian Computer Education Conference (ACE 2013), Adelaide, South Australia, January-February 2013. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 136, Angela Carbone and Jacqueline Whalley, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

ducting the research, with details on the unit under investigation, the composition of the student cohort, and the analysis method used. Results from the data collection phase of the research are then presented. After the Results section, the Discussion presents our interpretation and analysis of the data, including recommendations to help inform the development of units delivered using this approach, and ideas for future research in this area.

2 Method

This section is divided into three parts to clearly describe the *Introductory Programming Unit*, the *Student Cohort and Research Participation*, and the *Thematic Analysis of Reflections*. In the Introductory Programming Unit section we provide details of the unit that was investigated as part of this research. The Student Cohort and Research Participation section details the student body undertaking this unit and how they were recruited to be part of this research. Finally the Thematic Analysis of Reflections section outlines the process followed to extract and analyse the data from the student portfolios.

2.1 Introductory Programming Unit

The unit investigated in this work was a first year, first semester, programming unit. The design, development and delivery of this unit followed the principles we outlined in Cain & Woodward (2012). This involved the definition of *Intended Learning Outcomes*, supporting *Teaching and Learning Activities* and *Portfolio Assessment* with *Iterative Feedback*. Each of these are discussed in the following sections as context for the thematic analysis.

2.1.1 Intended Learning Outcomes

The Intended Learning Outcomes (ILOs), listed in Fig. 1, were central to all aspects of the unit. They formed the central focus for students, who were required to prepare a portfolio to demonstrate they had met these outcomes by the end of the unit. The ILOs guided teaching and learning activities, which were designed to help students build skills and to give them opportunities to develop work that could be included in their portfolios.

This unit aimed to introduce students to structured programming³ and this is reflected in the particular wording of the ILOs.

To allow students to explore these concepts a modern version of the Pascal programming language (Wirth 1971, Van Canneyt & Klämpfl 2011) was used, with a brief demonstration of the C programming language (Ritchie et al. 1978) toward the end of the unit.

2.1.2 Teaching and Learning Activities

Teaching activities took place in scheduled lectures and laboratory classes. The semester was thirteen weeks, twelve of which were teaching weeks, and a single week semester break in week eight. Topics for the twelve lectures are shown in the following list.

1. Programs, Procedure, Compiling and Syntax
2. User Input and Working with Data
3. Functions, Procedures, and Parameters
4. Branches and Loops
5. Custom Data Types

6. Functional Decomposition
7. Case Study
8. Pointers and Dynamic Memory Management
9. Structured Programming
10. Recursion and Backtracking
11. Portfolio Preparation
12. Review and Future Studies

The unit's delivery included an early introduction topic of "understanding syntax", where students were taught how to read programming language syntax using the visual "railroad" diagram syntax notation (Braz 1990). This allowed later lecture topics to focus on concepts, with syntax being offloaded to programming demonstrations and supplied notes, which included railroad diagrams and small code examples for each programming statement.

Allocated classes were designed with the goal of actively engaging students. Lectures typically included a review of previous topics, a short presentation using "Beyond Bullet Points" style lecture slides (Atkinson 2007), an interactive programming demonstration, and group activities. Laboratory sessions involved code reading activities, guided coding activities, and practical hands-on exercises.

2.1.3 Iterative Formative Feedback

Assessment in the unit included both formative and summative forms. Weekly assignments were submitted for formative feedback, with summative assessment of a portfolio submitted in the two week examination period that followed the thirteen week semester. A student's final grade was determined using criterion referenced assessment (Biggs 1996).

A high importance was placed on the iterative, formative feedback aspect of the portfolio assessment process (Cain & Woodward 2012). In this unit, students were required to submit their attempts of the weekly assignments at the start of each lecture. These were collected and marked by the unit's tutors, with the lectures being scheduled so that these exercises could then be returned to the students that same week. This enabled students to benefit from the feedback in a timely manner.

To enable the short timeframe between submission and return, tutors were instructed to focus on key issues, rather than all issues, apparent in the submitted work. This was intended to ensure that students received feedback that was relevant to them, and focused their attention on the most important areas they needed to improve.

Weekly assignments were submitted on paper, rather than electronically, to permit the tutors to rapidly review the documents and to encourage students to submit something of substance. Tutors quickly scanned through each submission looking for good qualities and issues to raise with the student. Sections of the code and answers were highlighted and then discussed directly with the student when their assignment was returned to them in the laboratory session.

No marks were allocated to these weekly assignments. They provided an opportunity for students to develop work to include in their final portfolio. This process allowed students to make mistakes without fear of losing marks from their final grade. It was expected that students would improve on their earlier

1. Read, interpret, and describe the purpose of sample code, and locate within this code errors in syntax, logic, style and/or good practice.
2. Describe the syntactical elements of the programming language used, and how these relate to programs created with this language.
3. Write small programs using the language provided that include the use of arrays, pointers, records, functions and procedures, and parameter passing with call by reference and call by value.
4. Use functional decomposition to break a problem down functionally, represent the resulting structure diagrammatically, and implement the structure in code as functions and procedures.
5. Describe the principles of structured programming and how they relate to the structure and construction of programs.

Figure 1: Intended Learning Outcomes for the unit investigated

Pass			Credit			Distinction			High Distinction		
P	P	P	C	C	C	D	D	D	HD	HD	HD
50	53	56	65	68	70	75	78	80	90	95	100
Portfolio includes: <ul style="list-style-type: none"> ▪ Learning Summary Report. ▪ Weekly assignment work. ▪ The three tests. All tests have been passed. Weekly assignments have been completed satisfactorily. Learning Summary Report includes a reflection on what you have learnt, and clearly shows how all intended learning outcomes are addressed to at least an adequate level .			In addition to meeting the Pass requirements the Portfolio includes: <ul style="list-style-type: none"> ▪ Bonus work from the weekly assignments. Bonus work is sufficient so that at least one of the intended learning outcomes is addressed at a good level .			In addition to meeting the Credit requirements the Portfolio includes: <ul style="list-style-type: none"> ▪ Additional bonus work. ▪ A program of your own design and implementation. ▪ Experience Report on the above program. Additional bonus work is sufficient so that in total at least two of the intended learning outcomes are addressed at a good level . The program and Experience Report are sufficient so that at least one of the intended learning outcomes is addressed at an outstanding level .			In addition to meeting the Distinction requirements the Portfolio includes: <ul style="list-style-type: none"> ▪ Additional bonus work. ▪ A research report. Additional bonus work is sufficient so that in total at least three of the intended learning outcomes are addressed at a good level . The program and Experience Report are sufficient so that in total at least two of the intended learning outcomes is addressed at an outstanding level . The research report is sufficient so that at least one of the intended learning outcomes is addressed at an exemplary level .		

Figure 2: Overview of assessment criteria provided to students in the unit outline.

Intended Learning Outcome: Write small programs using the language provided that include the use of pointers, records, functions and procedures, and parameter passing with call by reference and call by value.			
Adequate	Good	Outstanding	Exemplary
Pieces include programs that show the use of the following: <ul style="list-style-type: none"> ▪ Procedures to perform tasks. ▪ Functions to calculate and return values. ▪ Global and local variables. ▪ Command line arguments. ▪ Parameters passed by reference and by value. ▪ Simple statements including Assignment statements and Procedure Calls ▪ Branching statements including If and Case statements. ▪ Looping statements including While, <u>Until</u> and For statements. ▪ Arrays ▪ Custom data types including Records and Enumerations. ▪ Pointers, memory allocation and management. ▪ Code from a range of libraries. ▪ Code divided into multiple units. ▪ Recursive functions and data structures. 	Evidence <u>also</u> includes completed versions of most of the bonus programs from the programming assignments.	Evidence <u>also</u> includes a program of reasonable size and complexity that you have proposed, designed, implemented and tested. Accompanying the program is an Experience Report that discusses topics related to this ILO. Specifically: <ul style="list-style-type: none"> ▪ How you have applied your understanding of programming structures and abstractions to the program of your creation. ▪ The different programming language structures used in the creation of the program. ▪ How these structures helped shape the design of the program. 	Evidence <u>also</u> includes a Research Report related to programming structures such as: <ul style="list-style-type: none"> ▪ A report that discusses pointers, program memory layout, and memory management. ▪ <u>or</u> ▪ A report on how function and procedure calls work below the surface. ▪ <u>or</u> ▪ Other research topic as agreed with the convenor.

Figure 3: Example assessment criteria related to a single intended learning outcome.

submissions and include these versions in their final portfolio submissions.

Students sat three tests during the semester, with additional classes being scheduled so that they did not take away from the teaching and learning time. As with the weekly assignments, these tests carried no marks. The first two tests were formative, with students needing to get near-perfect answers but being permitted to correct issues once they received their papers back. This permitted the teaching staff to gain a better understanding of student progress, and helped students practice completing programming exercises in exam conditions. The final test had to be completed satisfactorily in exam conditions for the student to be eligible to pass the unit. Students who did not complete the test satisfactorily the first time were able to re-sit the following week.

2.1.4 Portfolio Assessment

Student portfolios contained a combination of the student's test work, work they had prepared in response to the weekly Assignments, a Learning Summary Report, and other pieces. The portfolio requirements and assessment criteria were included in the Unit Outline, and these were discussed with students in multiple lectures.

To be eligible for a Pass grade the portfolio had to include a range of pieces from the weekly exercises. The Credit grade required completion of extension exercises, including additional programs or reports on related concepts. Distinction and High Distinction required students to go beyond the set work: Distinction was awarded for portfolios that included a custom program of the student's design and creation, and High Distinction required a research report that analysed some aspect related to programming.

The overview of the assessment criteria from the unit outline, provided to the students in the first week, is shown in Fig. 2. Each of the ILOs then included a separate set of criteria on the different levels to which these outcomes could be demonstrated. An example is shown in Fig. 3.

Students were asked to reflect on their learning in the Learning Summary Report, and a template document was provided to assist students in preparing their comments. The template prompted students to describe the pieces they had included, to describe how these related to the ILOs, and then to reflect on what they had learnt from the unit.

To help students in writing their reflections, the following instructions were provided in the template.

Think about what you have learnt in this unit, and reflect on what you think were key learning points or incidents. Answer questions such as: What did you learn? What do you think was important? What did you find interesting? What have you learnt that will be valuable for you in the future? Which activities helped you most? Has this changed the way you think about software development? Did you learn what you wanted/expected to learn? Did you make effective use of your time? How could you improve your approach to learning in the future? Etc.

Note that there were no prompts for students to include details on issues they had encountered, meaning that any issues expressed should have been significant to the learning experience of the student.

2.2 Student Cohort and Research Participation

This unit was undertaken by 84 students, 70 of whom submitted a portfolio for assessment. Participation in

the research was voluntary, with informed consent being sought in lecture 11. All students who attended the lecture were required to fill in and sign the consent form, where they could indicate if they were willing to participate. To avoid any concerns regarding coercion, these forms were collected by a staff member not involved in the assessment of the unit, and stored until after unit results had been published. Students were made aware of these arrangements prior to giving consent.

Table 1 shows the number of portfolios made available to this research, the number that included comments related to the theme of "issues" and the distribution of grades. The grade distribution is also shown in Fig. 4, and will be discussed in Section 4.

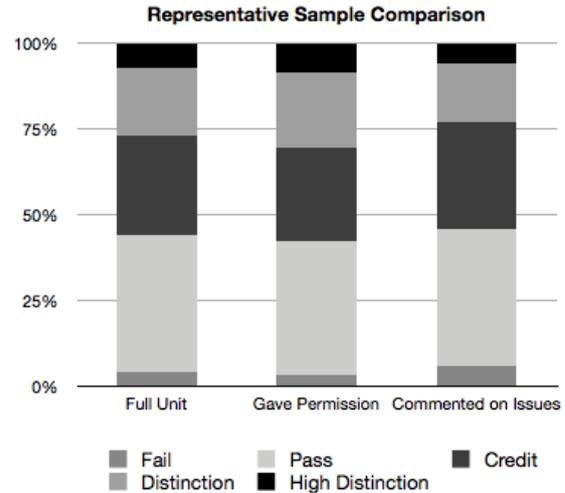


Figure 4: Distribution of grades for the full unit, for those students who agreed to participate in the research, and for those who commented on issues.

2.3 Thematic Analysis of Reflections

Reflections in student portfolios provide a wealth of information. To help identify the themes and patterns in these portfolios it was decided to perform a thematic analysis using the process outlined by Braun & Clarke (2008). This process involves six phases (with some terminology adapted for clarity):

1. Familiarising yourself with the data
2. Generating initial themes,
3. Searching for strong themes
4. Reviewing themes
5. Defining and naming themes
6. Producing the report

Familiarity with the data was obtained early in the process, with all of the portfolios being read as part of the unit assessment. At the end of the unit assessment notes were made in relation to the general issues that were raised in the portfolios and portfolio interviews.

Once the portfolios were made available for this research the initial themes were generated by revisiting the reflective component of each portfolio and looking for all explicit mention of issues the student faced.

Table 1: Portfolios submitted, issue comments and grade distribution.

	Total	HD	D	C	P	F
Submitted Portfolio	70	5	14	20	28	3
Agreed to participate	59	5	13	16	23	2
Commented on Issues	35	2	6	11	14	2
- Learning Issues	26	2	3	9	12	1
- Programming Issues	22	1	4	8	7	2

Each new issue identified was matched to a theme and recorded in a spreadsheet. The spreadsheet software was used to collate the themes and record the portfolio details of where these issues had been mentioned, along with any illustrative comments using the students own words.

In phases 3 through 5 the codes were grouped based on broader themes, and then into sub-themes. To ensure that all issues were reported in the results, the process we followed did not remove or ignore any issues raised. All issues that could not be grouped into an existing theme were collected together as a miscellaneous “other” theme. The Results section outlines the different themes identified, and how these themes relate to the comments raised by students in their reflections.

In the reporting of this analysis we present the raw coded results, grouped into the identified themes. Illustrative quotes from the student reflections are provided to help define the themes. Additional supporting evidence is also taken from the reflections of the teaching staff.

3 Results

A number of themes emerged from the analysis, and can be broadly classified as either general *learning issues* or *programming related issues*. (See Table 1 and Fig. 5.) Each of these categories is presented in Table 2 along with the number of students who raised these issues, broken down by grade. The following sections describe the individual themes in more detail.

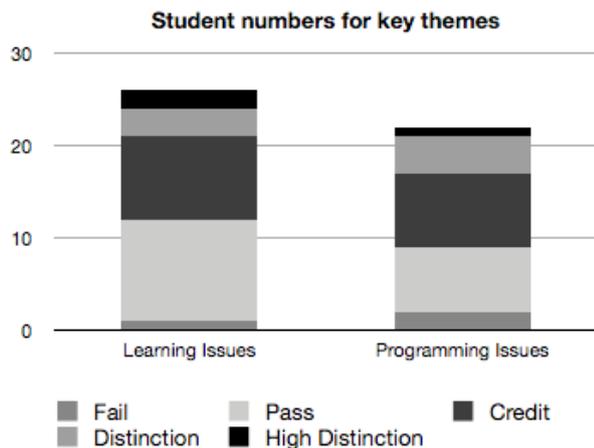


Figure 5: Number of students mentioning learning issues and programming issues. See Table 1.

3.1 General Learning Issues

The *general learning issues* capture all of the comments made by students that do not relate directly to

a given programming topic or technical aspect of the unit, but instead relate to the students’ learning experience in general. In this category the themes that appeared include *time management*, *getting started* with the unit, and *learning through mistakes*. The issue counts and grade distribution of these are included in Table 2, and can also be seen in Fig. 6.

Time management issues identified in the students’ reflections included comments about aspects such as “staying on task”, wishing they had “asked for help earlier”, or the general need to improve their time management to enable them to achieve higher grades. It can be seen that the majority of these concerns were raised by students who obtained either a Pass or Credit grade. These comments are further supported by observations from teaching staff, who noted concerns about students not working consistently through the semester and not seeking help in a timely manner.

The next largest general learning issue was *getting started* with programming. These comments specifically indicated issues related to the initial hurdle of getting started with the unit. One student noted this as their first experience using a computer, while others commented on the difficulty of the first few weeks’ lab exercises. Again, these findings are supported by observations from the teaching staff who noted that a large number of students withdraw from the unit before census date,⁴ and there was a general drop in enrolment numbers around this time. This may indicate that a larger number of students faced these issues but did not continue with the unit, though further work is needed to verify this.

The last main issue in this section related to students reflecting on the mistakes or struggles that provided them with an opportunity to learn something important, referred to as *learning through mistakes*. For example, one student’s reflection noted that:

“... I suddenly gained insight [into the code] I had been struggling with ...”

The reflection continued on to comment that having overcoming these issues the student gained a clearer understanding of the concepts taught up to that point, and that subsequent programs were easier to understand.

A number of *other* issues were identified by individual students. These issues included:

- transitioning to university life and study,
- finding information in the online learning management system,
- seeking help in general,
- keeping up with the pace of the unit, noted as “challenging but good”, and
- adjusting to portfolio assessment.

Table 2: Issue count results for grade and theme. Values of interest are indicated using bold format.

Theme	Description	Total	HD	D	C	P	F
Learning Issues	Issues related to learning in general.						
- Time Issues	Time constraints, or issues with time management.	14	1	0	5	8	0
- Getting Started	Comments relating to initial weeks, or tacking early hurdles.	8	0	1	2	5	0
- Learn through mistakes	Specifically commented on having issues and learning from these.	7	1	2	2	2	0
- Other	Other learning related concepts not allocated to other themes.	5	0	0	2	2	1
Totals			2	3	11	17	1
Programming Issues	Issues related to programming topics, or technical areas.						
- Pointers	Use of pointers and dynamic memory allocation functions.	11	1	2	4	3	1
- Parameters	Mentions parameters, or parameter passing	8	0	1	4	3	0
- Program Design	Algorithm and program structure design	7	0	1	1	3	2
- Other (Syntax)	Other issues, but related to the language syntax or concepts.	7	0	0	2	3	2
- Other (General)	Other programming issues not allocated to other themes.	5	0	0	2	2	1
- Recursion	Declaration and use of recursive functions or data structures.	4	0	1	2	1	0
Totals			1	4	14	11	3

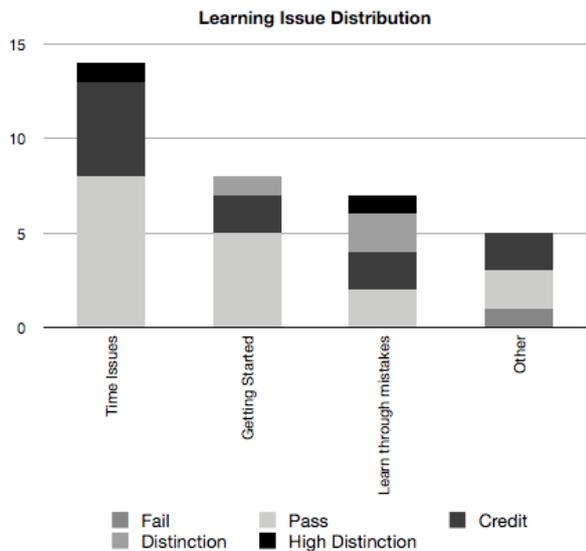


Figure 6: Number of students mentioning issues related to learning. See Table 2.

3.2 Programming Issues

As already mentioned, fewer students commented on programming or technical issues in their reflections than the more general learning issues. The programming sub-themes matched specific topics covered in the unit, including *pointers*, *parameters*, *program design*, and *recursion*. In this theme the *other* sub-theme featured more prominently, with a larger range of issues being located in the reflections of only one or two students. The data for these themes is listed in Table 2 and shown in Fig. 7.

Amongst the identified programming issues, *pointers* featured most prominently. Comments typically just referred to having issues with “pointers”, with the more detailed comments discussing issues with knowing when to dereference pointers and being unsure of when to use pointers. This is further supported by notes from teaching staff indicating that pointers tended to be problematic even for students who demonstrated strong programming skills up to this point in the material.

Parameters were also mentioned by a number of students as being a topic that was particularly challenging. This included comments relating to tracing parameter values through a number of function or

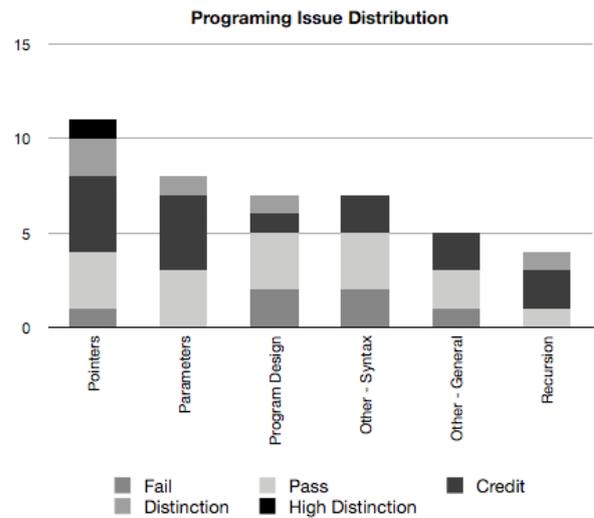


Figure 7: Number of students mentioning issues related to programming. See Table 2.

procedure calls, and issues of a single value having different names across different routines. From these comments there is a direct connection from parameter issues to a student’s understanding of program structure, or more importantly execution flow.

Issues relating to *Program Design* were also raised in the portfolio reflections. These comments related to aspects such as using functional decomposition, planning program structure, and designing algorithms.

The **other** issues for the programming category captures issues identified by one or two students. These were classified as relating either to **syntax and concepts** or **general programming** issues, and were:

- Syntax issues included:
 - iteration and working with loops,
 - using arrays (two comments),
 - creating composite data types using records,
 - functions in general,
 - dealing with syntax errors, and
 - using units to divide programs into multiple files.

- General programming issues included:
 - “Programming in general”,
 - “Following program code” in code reading exercises,
 - difficulties finding and using resources from the provided⁵ API, and
 - the math needed to achieve programming tasks.

There were also a number of reflections that raised the topic of **recursion**; these mentioned issues with both recursive functions and data structures.

4 Discussion

4.1 Investigation Focus and Sample Quality

Comments provided by students, when reflecting on their learning during any unit, can be valuable and interesting in many ways, especially with respect to the evaluation of a particular approach to teaching. Our investigation focused specifically on the theme of issues mentioned or identified by students in their reflective reports. Results of the thematic analysis, presented in Section 3, identified clear key themes. Additionally, several individual comments were selected.

The analysis considered a sample of reflective reports presented in a single semester unit. Of the 70 students in the class, almost 85% were willing to participate. Within the participant group, 35 students wrote one or more comments that matched the target theme. Table 1 and Fig. 4 show that the relative distribution of grades in the contributing group matches closely to both the participant group and the entire results for the unit. This strongly supports that the results are a representative sample of the unit, at least with respect to grade distribution.

4.2 General Learning Versus Programming Issues

Beginning with the two key themes of general learning issues and programming issues (Table 1 and Fig. 5) it can be seen that the distribution of student grades is very similar, with a slightly stronger representation of Pass students in the learning issues theme.

Overall, more students commented on learning in general. This is of particular interest given the relative emphasis of the course material, which focuses on teaching programming concepts over syntax details. Despite the relatively small time spent on syntax, students did not mention having related issues.

A closer examination of the issues related to programming strengthens this analysis further. Most student comments on programming issues (Table 2) concerned applying programming concepts, rather than issues of understanding syntax. Also, these comments were about when and how to use the related programming concepts rather than specifically how to apply the syntax of the language used.

Comparison of the grade distributions within the learning issues (Fig. 6) and programming issues (Fig. 7) suggests potentially interesting differences, such as issues specific to grade groups, and other issues across all grades. The sample size of this investigation limits any significant insight although some points are listed in later discussion.

4.3 Learning Issues

4.3.1 Time Management

Time management issues were identified by the largest number of students (Table 2). The grade distribution is skewed towards student’s who achieved Pass and Credit results (bold values), suggesting that students who do achieve Distinction or High Distinction results managed time better, and that the unit structure requires good time management to achieve these outcomes.

Developing a portfolio that demonstrates the ability to apply concepts taught requires time: time to practice using the concepts, and time to demonstrate their use competently. For students to achieve Distinction and High Distinction grades, they need to be able to organise their time effectively.

With more traditional forms of assessment, marks can be used as incentives. Using assessment due dates during the delivery of the unit has the effect of turning marks into time distributed weighted incentives. Marks no longer represent the importance of the learning outcome, but match allocation of incentive. Consider, for example, the allocation of marks for lab attendance. These marks do not help measure the students’ learning outcomes, but are purely there to incentivise lab attendance. Similarly, assignments due within the unit delivery period assess the speed of acquiring the required knowledge.

With portfolio assessment the summative assessment is delayed until after unit delivery. This has the benefit of providing a more direct assessment of learning outcomes, but has a cost related to loss of incentives during delivery. While this is positive from a learning perspective, it can easily lead to students delaying their work on portfolio assessed units in order to address the more time critical assignments in other units. Given the number of comments related to this issue, it appears to be easy for students to then lose sight of how they are falling behind in a unit with a relatively flexible portfolio assessment.

4.3.2 Getting Started

Getting started is another issue facing many students (Table 2 and Fig. 6). In the first few weeks of the semester, students will face practical and conceptual issues. Practical issues include installing compilers and text editors, learning to use command line tools, and issues with general computer use. At the same time students need to build a viable conceptual model of computing (Hoc & Nguyen-Xuan 1990), and relate this to the programs they are creating.

Early on students may also face challenges transitioning to university study and university life in general. In the first few weeks students are also more likely to have issues with syntax, and dealing with syntax errors. Together these challenges can present a significant hurdle for students.

These issues could be addressed in a number of ways. Shifting toward an IDE could remove some issues related to the use of the command line compiler, but add overhead related to use of a more complex programming environment, and do not assist students in building their conceptual model of computing. The teaching staff also felt that students undertaking this unit do need to learn to use the command line, and this early introduction meant that later units could expect at least some familiarity with command line tools.

4.3.3 Learning Through Mistakes

The students' active role in building their own conceptual model of a topic plays a significant role in constructive learning theories (Glaserfeld 1989). Effective teaching then becomes the ability to place students in situations where errors in their understanding can be challenged to help the students build viable conceptual models.

With this in mind, it is interesting to note, as shown in Table 2, the number of students who commented on gaining significant understanding through making mistakes. In line with constructive thinking, these students encountered situations in which their conceptual model was inappropriate, and in addressing the associated problems they were able to gain a better, more robust, conceptual model.

Comments about learning through mistakes were distributed across all grades, from Pass through to High Distinction (Fig. 6). This suggests that mistake-based learning experiences are beneficial to a wide range of students, albeit with some gaining a better understanding than others through the process.

4.3.4 Other Learning Issues

From the other issues students noted, many can be attributed to transitioning to university education. Learning to locate and use learning resources and to seek help, are all issues that students must come to deal with when shifting to university education.

It is interesting to note that one student did raise a complaint about portfolio assessment, indicating that it would be easier to sit an exam. While this is only a single student, it does highlight that the purpose of the ongoing assessment may not be realised by all. Tang et al. (1999) indicated that students tend to apply narrower learning strategies for examinations, focusing on memorising material covered in lectures. In contrast, Tang et al. (1999) found that with portfolio assessment students adopted a wider perspective, making use of higher cognitive activities such as application, relation, and reflection. Students are likely to find these higher cognitive activities more challenging, and therefore those who wish to apply surface learning approaches are likely to prefer other assessment strategies.

4.4 Programming Issues

4.4.1 Pointers and Recursion

Our results support those from Lahtinen et al. (2005) in indicating that students find learning pointers challenging. Issues related to using pointers and memory management featured across a range of grade results (Table 2 and Fig. 7), indicating that this concept was challenging even for those students who managed to achieve good results in the unit.

Pointers require a good conceptual understanding of computing, and the ability to debug logical errors. Issues with pointers can often result in abrupt program termination, which can be very confronting for beginner programmers. Locating the cause of these errors is an additional challenge, that requires a student to build a mental model of what is happening within the programs they have written.

Issues with recursion were raised by fewer students than other issues, which is in contrast to the study by Lahtinen et al. (2005). This may be explained by the short time students had with recursion in this study. A deep exploration of recursion was not required for students to pass the unit. It is likely, therefore, that

many students may not have had sufficient time to explore more complex applications of recursion.

In addition to being complex, pointers and recursion both occur relatively late in the curriculum. With pointers, students had little time to develop the skills necessary to handle associated issues, whereas with recursion the short time meant students had little opportunity to develop programs of sufficient complexity to encounter issues. In either case, at the time of writing their reflections, issues with later lecture topics are perhaps more likely to be in focus.

4.4.2 Parameters

Parameters require students to understand local scoping of variables, procedure and function calls, and methods for sharing these values between functions and procedures. This appears to be another point at which students need to expand their model of computing (Hoc & Nguyen-Xuan 1990).

While parameter concepts can take time to understand, issues are likely to be constructive in nature. When the logic for a program is contained within a single procedure, students can develop a simplistic model of what is occurring when other functions or procedures are called. When students need to design their own functions and procedures that require parameters, they are presented with situations that challenge their simplistic model. This suggests that parameters provide a significant learning opportunity from a constructive perspective.

The two different parameter passing methods are both taught in the unit, with pass by reference being used to create procedures to swap parameter values, as well as allowing procedures to modify data within structures and arrays. Call by reference provides an early introduction to references.

4.4.3 Program and Algorithm Design

Program and algorithm design are progressively taught throughout the unit, with the main focus being in the middle of the unit's delivery in topics related to functional decomposition and structured programming. Comments by students indicated several issues on how to practically apply the concepts covered to create programs.

The authors of this paper initially expected a larger representation of this issue, as design tasks require a deeper, relational, understanding of the concepts being used. However, the core tasks students had to submit for a Pass grade were accompanied with detailed instructions to help ease these design issues. Extension tasks required for a Credit grade did require some design components, and less guidance was provided. Students attempting their own program, necessary for a Distinction grade, needed to perform design activities as these programs were of their own design and creation.

4.4.4 Other Programming Issues

The other programming issues raised by students can be classified as individual challenges. It seems that students are likely to learn at different paces, in different ways, and find different topics challenging. Again, general comments concerned the application of programming concepts, rather than with basic syntax. Each of the raised issues indicated a point at which students had an opportunity to challenge and develop their conceptual understanding of programming and their model of computing.

4.5 Recommendations

Based on the thematic results and on the experiences of staff involved in the unit delivery, there are a number of implications and recommendations that can be made. These recommendations are listed below, and will be explained in later sections:

- strongly avoid mixing formative with summative assessment,
- give students time to adjust to portfolio assessment,
- focus on student “awareness”,
- use a quick formative feedback process,
- avoid the “tutor debugging” phenomena,
- use visual methods to convey progress, and
- make students aware of issues they are likely to face.

4.5.1 Always formative, lastly summative

Separating formative feedback processes from summative marking has a clear value, and this is reflected in student comments. Our observation is that using a punitive marking system creates an incentive for students to hide faults and limits in their understanding. Students need to know what they need to learn. Related to this is the time a student can spend asking about marking schemes or lost marks – time better spent on learning.

4.5.2 Students need time to adjust

In comments to staff, students have said that it takes time to get used to a portfolio based unit even if they understand the principles. If we consider that students might be conditioned to respond to summative marking and due dates as a way of allocating their attention, an interesting question emerges: how do we help students maintain an active engagement with the unit activities? Finding an answer for this is an ongoing challenge and research opportunity.

4.5.3 Focus on student awareness

Primarily, student awareness is the basis for positive engagement and an aware student has the opportunity to make appropriate choices. To support this, staff need to communicate the structure, activities and expectations of a portfolio-based unit to students as effectively as possible. Unfortunately students may essentially have habits that can take time to adjust. It is possible to help students with issues such as time management and, hence, learning outcomes.

Although formative activities many not have due date or marks (grade penalties), staff should still express clear expectations of when work needs to be done. In some cases this leverages a students’ habits to their advantage as they feel compelled to do the work. Ideally, students should give these formative tasks as high a priority as assignments with marks.

4.5.4 Use quick formative feedback

Very quick feedback helps to create strong reinforcement in a student that the process really is formative and personally valuable. In a students’ experience summative marking is often a delayed process. If formative feedback takes a long time it is removed

from the students’ current learning and challenges, and so can be confused as summative marking. Students need to be engaged with the formative nature of these assessments, making use of the feedback to help develop their understanding.

4.5.5 Avoid tutor debugging

A possible problem with quick formative feedback, and resubmission opportunities, is that students may submit poorly prepared “drafts” and use staff simply to “fix things”. This issue has been described as “tutor debugging” by some of our staff, and should be actively discouraged. One approach to this is to set minimum submission standards for work submitted for feedback.

4.5.6 Use visual methods to convey progress

Visual charting of tasks and completed work, calendar events and strong reminders of work and time limitations help to engage students. It is also possible that a “gamefication” approach, by recognising personal or group achievements and rewarding with awards, “badges” an other game-related concepts, can create a fun and personal incentive for students. We also recognise that there are also risks with gamefication, such as trivialisation of the value of core learning activities or distorting the value of learning activities through association to a gamefication artefact.

4.5.7 Tell students what to expect

Finally, helping students understand the issues they are likely to face should help them prepare sufficiently for the more challenging tasks. This is particularly relevant to the issues related to getting started. The challenges early on in the unit may put a number of students off, and these students are likely to lose motivation and engagement with the unit. Making them aware that these challenges are “normal”, and to be expected, may assist them in getting over early hurdles.

4.6 Future Work

It would be valuable to compare the thematic results between different semester groups undertaking the same unit of study to see if the results are similar. We would also like to compare results from this programming unit to later programming units. Similarly, as programming units are undertaken by students from different courses, each with different expected aims and outcomes, it would be interesting to see if the themes identified correlated to particular groups. Ideally this could inform both the development and delivery of programming units, as well as courses as a whole.

Once students are familiar with the learning environment, and expectations of a unit delivered using constructive alignment and portfolio assessment, it would be reasonable to expect some indication of this in the reflection comments presented by students. Future investigations could look for changes in themes such as *getting started* or *time management* which would ideally improve in students with experience.

While this investigation focused on the issues identified by students, there are other themes that could be used as a focus for thematic analysis. Themes could be compared to the learning modes or preferences of students, which we would expect to strongly correlate to reflection themes in some cases.

5 Conclusion

In this paper we have presented a thematic analysis of reflective reports presented by students as part of their assessment in an introductory programming unit. The development and delivery of the unit was described in detail as a context for the work. A good representation of students distributed across all result grades agreed to participate in the study.

Thematic analysis was directed specifically at the theme of *issues* identified by students. Overall results showed that more students raised learning issues than programming related issues. Significant learning themes included time management, getting started and mistake-based learning. The most common programming issues were related to pointers and parameters, with only a small number of issues related to syntax, and both these results were expected. Issues related to program design were raised less than expected.

The discussion considered a number of interesting results, and put forward recommendations and future directions for research in this area.

Notes

¹Unit in this context refers to a course/subject/module within a degree programme.

²This research was granted ethics approval in accordance with [institution details anonymised].

³Structured programming in the context of the ILOs has a broad meaning, encompassing imperative, procedural, and structured design concepts, as well as control flow.

⁴This is the date when the university records enrolment numbers, typically a few weeks after the start of the semester to allow for changes of enrolment.

⁵Students were provided with an API for creating small games. This included functionality for drawing shapes and images, playing sound effects and music, handling input, and other functions and procedures related to creating small 2D games.

References

- Armarego, J. (2009), Constructive alignment in se education: aligning to what?, in H. Ellis, S. Demurjian & Naveda, eds, 'Software Engineering: effective teaching and learning approaches and practices', ACM, pp. 15–37.
- Atkinson, C. (2007), *Beyond bullets points: using microsoft®office powerpoint®2007 to create presentations that inform, motivate, and inspire*, Microsoft Press.
- Biggs, J. (1996), 'Enhancing teaching through constructive alignment', *Higher Education* **32**, 347–364.
- Biggs, J. & Tang, C. (1997), 'Assessment by portfolio: Constructing learning and designing teaching', *Research and Development in Higher Education* pp. 79–87.
- Braun, V. & Clarke, V. (2008), 'Using thematic analysis in psychology', *Qualitative Research in Psychology* **3**(2), 77101.
- Braz, L. M. (1990), Visual syntax diagrams for programming language statements, in 'Proceedings of the 8th annual international conference on Systems documentation', SIGDOC '90, ACM, New York, NY, USA, pp. 23–27.
URL: <http://doi.acm.org/10.1145/97426.97987>
- Cain, A. & Woodward, C. J. (2012), Toward constructive alignment with portfolio assessment for introductory programming, in 'Proceedings of the first IEEE International Conference on Teaching, Assessment and Learning for Engineering', IEEE, pp. 345–350.
- Cohen, S. A. (1987), 'Instructional alignment: Searching for a magic bullet', *Educational Researcher* **16**(8), 16–20.
- Glaserfeld, E. (1989), 'Cognition, construction of knowledge, and teaching', *Synthese* **80**, 121–140.
URL: <http://dx.doi.org/10.1007/BF00869951>
- Hoc, J. M. & Nguyen-Xuan, A. (1990), 'Language semantics, mental models and analogy', *Psychology of programming* **10**, 139–156.
- Lahtinen, E., Ala-Mutka, K. & Järvinen, H. M. (2005), 'A study of the difficulties of novice programmers', *ACM SIGCSE Bulletin* **37**(3), 14–18.
- Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M. & Paterson, J. (2007), 'A survey of literature on the teaching of introductory programming', *ACM SIGCSE Bulletin* **39**(4), 204–223.
- Ritchie, D. M., Johnson, S. C., Lesk, M. E. & Kernighan, B. W. (1978), 'The c programming language', *Bell Sys. Tech. J* **57**, 1991–2019.
- Robins, A., Rountree, J. & Rountree, N. (2003), 'Learning and teaching programming: A review and discussion', *Computer Science Education* **13**(2), 137–172.
- Tang, C., Lai, P., Arthur, D. & Leung, S. F. (1999), 'How do students prepare for traditional and portfolio assessment in a problem-based learning curriculum', *Themes and Variation in PBL. Newcastle: Australian Problem Based Learning Network*.
- Thota, N. & Whitfield, R. (2010), 'Holistic approach to learning and teaching introductory object-oriented programming', *Computer Science Education* **20**(2), 103–127.
- Van Canneyt, M. & Klämpfl, F. (2011), *Free Pascal User's Guide*, 2.6 edn.
URL: <ftp://ftp.freepascal.org/pub/fpc/docs-pdf/user.pdf>
- Wirth, N. (1971), 'The programming language pascal', *Acta informatica* **1**(1), 35–63.

Computational Thinking and Practice

— A Generic Approach to Computing in Danish High Schools

Michael E. Caspersen and Palle Nowack

Centre for Science Education, Faculty of Science and Technology

Aarhus University

DK-8000 Aarhus, Denmark

{mec, nowack}@cse.au.dk

Abstract

Internationally, there is a growing awareness on the necessity of providing relevant computing education in schools, particularly high schools. We present a new and generic approach to Computing in Danish High Schools based on a conceptual framework derived from ideas related to computational thinking. We present two main theses on which the subject is based, and we present the included knowledge areas and didactical design principles. Finally we summarize the status and future plans for the subject and related development projects.

Keywords: curriculum structure, course content, high school, computational thinking, core competencies, application areas, knowledge areas, learning activities, didactical design principles.

1 Introduction

Computing, particularly in the specific form of computer science, has been a topic in high schools in many countries for more than three decades, but without achieving the break-through in terms of adoption that the topic deserves in the post-industrial society.

But things are changing, and they are changing at a global scale. Internationally, there is a growing awareness on the necessity of providing relevant computing education in schools, particularly high schools. Computing education in schools is considered increasingly important as expressed by e.g. Wing (2006) who argues for teaching fundamental computing principles for all: “Computational thinking is a fundamental skill for everyone, not just for computer scientists. To reading, writing, and arithmetic, we should add computational thinking to every child’s analytical ability”. In the book *Program or be Programmed*, Rushkoff (2010) puts it even more bluntly: “In the emerging, highly programmed landscape ahead, you will either create the software or you will be the software”.

Half a century ago, Perlis (1962) said that everyone should learn to program as part of a liberal education. He argued that programming was an exploration of process, a topic that concerned everyone, and that the automated execution of process by machine was going to change

everything (Guzdial 2008). It took fifty years to get here, but finally it seems that (a contemporary interpretation of) Perlis’ vision has come to pass.

As mentioned by Cutts, Esper, and Simon (2011), several national initiatives are being taken to address this challenge. For example, the UK Royal Society has recently published the report *Computing in School* (Royal Society 2012), and the US National Science Foundation and the College Board are supporting development of an Advanced Placement course, CS Principles (Astrachan et al. 2012), aiming at broadening participation in computing and computer science by transforming high school computing (Astrachan et al. 2011). Similar initiatives are taken in other countries, e.g. Israel (Gal-Ezer and Harel 1998 and 1999, Bargury 2012), Germany (Steer and Hubwieser 2010), The Netherlands (Van Diepen et al. 2011), and Norway (Hadjerrouit 2009). Especially the effort in New Zealand seems to be similar with respect to motivation, and challenges, but perhaps not with respect to the content and form (Bell et al. 2010, Bell et al. 2012).

In this paper, we report on a recent Danish initiative to redefine and revitalise computing in Danish high schools. The Danish initiative is similar to many of the other initiatives in focusing on fundamental computing principles (including computational thinking) as a fundamental skill for all. However, the Danish initiative is different from most of the other initiatives in taking a broader and generic approach to computing rather than the traditional and narrower computer science or software engineering approach. This is a deliberate choice made primarily to embrace more fundamental aspects of computing (e.g. impact of information systems, the role of it in innovation, and interaction design for it-based systems), but also to accommodate the four different types of high schools in Denmark (general high schools, upper secondary shorter general education programme, technical high schools, and business high schools) with one generic computing subject.

In section two we briefly recap the history of computing in Danish High School curricula. Section 3 describes the two main theses that together define the perspective from which the new generic computing subject was designed. The subject is then fleshed out in the following two sections: Section 4 describes the knowledge areas of the subject, and Section 5 describes the didactical design principles behind the subject. Finally, Section 6 briefly summarizes the current status and plans for the subject.

2 Computing in Danish High School 1971-2011

Various flavours of computing has been a topic in Danish high school for more than forty years.

After some early individual initiatives in the late sixties with computing education in high schools, the *Johnsen Committee* was formed in 1971 to give recommendations regarding EDP education (Electronic Data Processing) in the Danish education system (Johnsen 1972). The recommendations of the Johnsen committee guided the computing curriculum decisions in general high school for more than ten years. However, the full set of recommendations —encompassing a mandatory computing subject for all high school students— were never implemented.

In 1980, the Ministry of Education published the so-called *Obel/Fisher Circular* recommending computing in general high school to be integrated in other subjects and phased-out as an independent subject. In the 1980s, computing remained an independent subject only in one branch (out of four) of Danish high schools. In 1987, computing became again an independent subject, but still only as an elective, and it has remained as such until today.

In business high schools, computing has been a subject since the mid-1980s —always with a special flavour of business, management, and administration.

From the mid-1990s and onward, other computing subjects saw the light of day in the different types of high schools, e.g. Information Technology, Programming, and Multimedia.

A major high school reform in 2005 dramatically reduced the conditions for elective subjects such as computing, and the same pattern emerged in all types of high schools: hardly any pupils chose computing and the subject almost completely vanished from the schools.

In late 2008, the Ministry of Education established a task force to conduct an analysis of computing in high schools and provide recommendations for a revitalisation of the subject (Agesen and Nørgaard 2009). The major recommendations of the task force were:

- To distinguish between computer literacy (emphasizing it-usage, e.g. the use of spreadsheets, word processing, and other applications) and computational thinking and practice (emphasizing creational and constructional competencies).
- To develop a single, coherent, and uniform computational thinking and practice subject, which then can be offered in several flavours.
- To design the course such that it may inspire pupils to continue with computing studies after high school.

The recommendations gained political support at all levels, and a new generic computing subject has been developed and is offered by volunteering schools for a three-year test period (2011-2014).

3 Foundational Theses

In general, young people do not consider computing a proper subject, and they certainly do not realise the importance and potential of computing in modern society. The main purpose of the new computing subject for high school is to convey the message condensed in the first of two foundational theses:

Thesis 1: *Through computing, people can create, share, and handle thoughts, processes, products and services that create new, effective, and boarder-crossing opportunities -impossible without the digital technology.*

The wording is a bit heavy, but the essence is quite similar to Wing's notion of computational thinking. Thesis 1 is the keynote of the new computing subject; as such, it must permeate all concrete learning activities that will be developed.

The second thesis relates to our ambition of embracing more fundamental aspects of computing but also to accommodate the four different types of high schools in Denmark with one generic computing subject. The thesis also reflects the diversity and various flavours of computing in academia, education, and industry.

Thesis 2: *There exists a common and shared foundational set of computational concepts, principles and practices, which can be applied purposefully within science & technology, business and social science, arts and humanities, and health and life sciences.*

Both theses were formulated before we commenced concrete development of the new computing subject. Throughout development, the theses served as guiding principles for our efforts of refinement and concrete design of the subject. In particular, thesis 2 provided guidelines for identification of seven core knowledge areas that has come to define the new computing subject. The seven knowledge areas are presented in the following section.

4 Knowledge Areas

We use the term “Knowledge Areas” in the same sense as in the curriculum recommendations from ACM¹: the areas are not to be thought of as *teachable* modules by themselves, but as appropriate categories for *describing* subject content. Hence, the categories are for description, and not didactical design of practical learning activities. We expand on these issues in Section 5.

In the following, we motivate and describe the seven knowledge areas that have been chosen for characterising the new computing subject and for formulating learning goals. The areas have been chosen after a short and intensive dialogue with selected colleagues from Danish universities. In retrospect, some of the areas are related to the computing practices suggested by (Denning 2003).

The knowledge areas are:

- Importance and Impact
- Application Architecture
- Digitisation
- Programming and Programmability
- Abstraction and Modelling
- Interaction Design
- Innovation

For each area, we provide a brief description and present the associated learning goals, as they appear in the formal curriculum. It should be noted, that the learning goals may appear overly ambitious, but they must of course be interpreted in the context of level, preconceptions, and allocated time for the actual course delivery.

¹<http://www.acm.org/education/curricula->

4.1 Importance and Impact

To truly understand and appreciate the importance of computing in modern society, the pupils must be presented to a portfolio of important and for the pupils relevant systems and innovations (e.g. Facebook, iTunes, GPS-based navigation systems, email, health care systems, etc.) — systems that the pupils know and can relate to. The design of an IT system has strong consequences for the people, organisations, and social systems that use it. Designers do not only design the system but also use patterns and workflows that unfold through the use of the system. The purpose is to make the pupils aware of the interplay between design of a system and the use patterns which the system intentionally or unintentionally generates.

Pupils must be able to

- Give examples of the impact of IT systems on human behaviour.
- Analyse and assess the importance and implications of IT systems and how they impact human behaviour.
- Apply user-oriented techniques for construction or modification of IT systems.

4.2 Application Architecture

The majority of IT systems are structured according to the so-called three-tier model consisting of a presentation tier, a logic tier, and a data tier. The model is relevant partly because it provides a general framework for understanding a very large class of IT systems, their components, and the interplay between these, and partly because the model is useful for qualified use of concrete systems, e.g. the Office package, Photoshop, iTunes, Facebook and general types of systems, e.g. simulation tools, accounting systems, content management systems, mobile technology, and computer games.

Pupils must be able to

- Describe principles for the architecture of IT systems.
- Apply specific architectures for construction of simple IT products and adjustment of existing IT systems.

4.3 Digitisation

In order to understand the basic characteristics of the computer, the pupils must understand and work with representation and manipulation of data. The main point is that data need to be digitised to allow representation in a computer and manipulation by programs. The purpose with this topic is that the pupils gain concrete experience with (and hence understanding of) representation and manipulation of data including the fact that digitising often results in loss of information. The other side of the coin is that digitisation and manipulation makes it possible to create new data. IT security is another important issue that may be addressed.

Pupils must be able to

- Describe the representation of selected types of data (e.g. images, sound, text, etc.) and construct IT products (programs) that make simple manipulations of data.

- Integrate various types of data in simple IT products and extend functionality of existing IT systems by adding new types of data.

4.4 Programming and Programmability

Computers are indeed very simple machines that gain their power through scale. The defining characteristic of the computer is its programmability and universality. Programming comes in many forms, but common to these is the principle of defining and hence automating computations that can be executed again and again with arbitrary data and data sets.

Pupils must be able to

- Identify basic structures in programming languages, construct IT products (simple programs) and adjust existing programs.
- Apply programming technologies for development of IT products and adjustment of existing IT systems.

4.5 Abstraction and Modelling

The purpose of this topic is to provide insight into modelling where data, processes and systems are described at an abstract level where design alternatives and properties can be evaluated and choices and decisions can be made.

Pupils must be able to

- Give examples of models of data, processes and systems and describe the relation between a concrete model and the relevant associated parts of an IT system.
- Implement selected models in a concrete IT product and adjust existing models and implement these adjustments in existing IT systems.

4.6 Interaction Design

The previous topic is primarily about models for elements of the presentation and logic tiers of the three-tier model. This topic is about models and design principles for the presentation tier — the interface where users and other systems meet an IT system. It's the purpose that the pupils understand the premises for as well as the consequences and importance of interaction design.

Pupils must be able to

- Describe and analyse selected elements of a user interface design, construct simple user interface designs and adjust existing designs.
- Implement selected interaction design in a concrete IT product and adjust existing designs and implement these adjustments in existing IT systems.

4.7 Innovation

The subject treats innovation from a product as well as process perspective. The subject takes an innovative approach to IT product development and provides a background for understanding aspects of IT product development and the interplay between IT and users/society.

Pupils must be able to:

- Characterise innovative development processes and sketch ideas for innovative IT products.

5 Didactical Design Principles

A number of didactical design principles and guidelines have been enforced, or at least highly recommended, for the development of learning materials for the new computing subject. In this section, we present the five major didactical principles:

- A learning activity is not (necessarily) the same as a knowledge area.
- Learning activities should be application-oriented.
- Learning activities should facilitate and guide a consume-before-produce progression through the materials.
- Learning activities should include several substantial worked examples.
- Learning activities should illustrate stepwise improvement as a general approach to incremental development of artefacts.

Other principles have been used such as game-based learning and narrative media-approaches (e.g. Andersen et al. 2003).

For a more general discussion of didactical approaches to computing, see Bennedsen et al. (2008) and Hazzan et al. (2011).

5.1 Knowledge Areas vs. Learning Activities

The knowledge areas introduced in Section 4 helps to structure the entire curriculum, but it is not a feasible structure for teaching the subject, as it would imply a sequential depth-first approach to the subject as a whole.

Instead we have adopted a well-known teaching strategy from Danish high schools, in which subject matter from various different knowledge areas are extracted and combined to piecemeal construct and deliver smaller packages of contextualised and interdependent subject matter components. These learning activities form the toolbox, from which the teacher select, combine, design, and implement his/her particular version of the subject which should be adapted and adjusted to the relevant context (education, level, and individual pupils). A learning activity may include subject matters from one, multiple, or all of the seven knowledge areas as illustrated in Figure 1. A learning activity is comprised by a description for pupils and teachers, materials and resources, and a process (cookbook) for using the materials in the learning activity.

The latter also illustrates a characteristic difference between knowledge areas and learning activities: the former are more static and are expected to change at a much slower pace than the learning activities, which are expected to change rapidly over the years, as technology and trends changes. Put another way: when the knowledge areas change, the whole identity of the subject changes (ranging from minor adjustments to radical changes in conceptual frameworks). Furthermore, changes in learning activities could be made for purely pedagogical reasons.

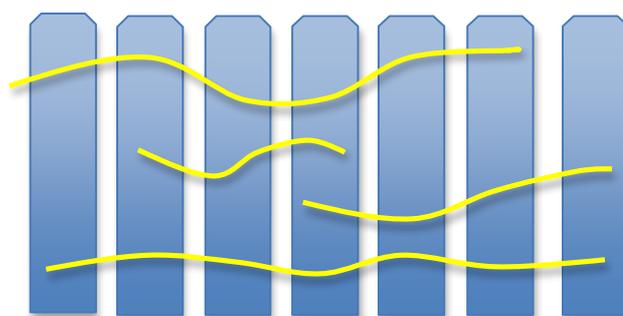


Figure 1: Content Structure Framework: Knowledge Areas (blue columns) versus Learning Activities (yellow lines)

5.2 Application-oriented (outside-in)

Traditionally, introductory computer science courses apply a bottom-up approach, in the sense that pupils are introduced to basic and foundational concepts and expected to master these before more advanced concepts and principles are introduced. Hence, in a traditional programming course, pupils are often trained in constructing a “Hello World” program as the very first activity, and then later on are trained in adding more layers of complexity to a system in terms of user interfaces, databases, etc. For the technically inclined pupils this may be a feasible approach, but in our case, this could pose severe motivational problems, as we are dealing with a wider range of pupils with much more diverse interests and backgrounds.

There is an even more important reason why a traditional bottom-up approach is fallible. We are not aiming at developing detailed and specific competences in the seven knowledge areas. Overall, we are aiming at developing interest, critical thinking, and broader skills in computational thinking and practice. Therefore we have decided on an application-oriented top-down approach. This means, that we start the various teaching activities by introducing well-known or familiar applications, which we then split apart for conceptual and/or technical examination, evaluation, and modification. For motivational reasons, we choose applications based on the criteria, that they must by themselves be naturally appealing to pupils in our age range. Applications, which they find interesting to use and hopefully to examine and improve. Examples could include pedagogical lightweight versions of Facebook, iTunes/Spotify, YouTube, Twitter, Blogs, Photoshop, and similar applications.

5.3 From Consumer to Producer

When designing learning activities, we aim at organising the material in such a way that the pupils experience a *consume-before-produce* progression through the material. Initially, the pupils act as consumers of an artefact by using and studying it; then, they go on to make first simple and then gradually more complex modifications to the artefact. Eventually, the pupils may be requested to build similar artefacts from scratch.

The *consume-before-produce* principle —sometimes alternatively characterised as a *use-modify-create* progression— can be applied in many areas. In programming, pupils can use programs or program modules be-

fore they start making modifications and eventually create modules or complete programs on their own. The approach applies equally well to other areas, e.g. modelling and interaction design.

The origin of (a specialisation of) this principle can be traced back at least to 1990 where Pattis introduced the *call-before-write* approach to teaching introductory programming (Pattis 1990). In Christensen and Caspersen (2002), the authors apply the principle to provide an alternative and incremental way of teaching about software frameworks and event-driven programming in CS1. In Schmolitzky (2005), the author briefly mentions the notion of consuming before producing by providing three specific examples of using the principle in the context of learning object-oriented programming using the BlueJ system (Kölling 2003).

5.4 Worked Examples

A Worked Example (WE), consisting of a problem statement and a procedure for solving the problem, is an instructional device that provides a problem solution for a learner to study (Atkinson et al. 2000, Chi et al. 1989, LeFevre and Dixon 1986). WEs are meant to illustrate how similar problems might be solved, and WEs are effective instructional tools in many programs, including computing.

Bennedsen and Caspersen (2004) illustrate implicitly how WEs can be used to teach object-oriented programming using a systematic, model-based programming process. Caspersen & Bennedsen (2007) present an instructional design for an introductory programming course based on thorough use of WE. Caspersen (2007) provides an overview of WE literature related to programming education as well as a survey of the related cognitive load theory.

Through didactical training of teachers and systematic enforcement, WE have come to play a key role in the didactical design of most learning activities developed for the new computing subject. A multitude of examples are available from a website maintained by the Danish Association of High School Teachers in Computing². Unfortunately, the material is only available in Danish.

5.5 Stepwise Improvement

The Danish Ministry of Education's official guidelines for the new computing subject recommend that all constructional activities be designed according to *Stepwise Improvement*. In its original form, stepwise improvement (not to be mixed with stepwise refinement although the two are somewhat related) is presented in the context of program development (Caspersen 2007, Caspersen and Kölling 2009), but the methodology is applicable for the construction of any concrete or abstract artefact.

Stepwise improvement is a framework for incremental development of an artefact. According to stepwise improvement, development takes place in three dimensions: from abstract to concrete, from partial to complete, and from unstructured to structured. Thus, development of an artefact can be characterised as a mixed sequence of refinements, extensions, and restructurings of the artefact.

For the new computing subject, the recommendation from the Ministry of Education is that stepwise improvement is used systematically in all constructive learning activities. A number of concrete examples as well as more general guidelines are provided in eight reports published by the Danish Ministry of Education (2011).

6 Summary, Status & Plans

In this paper we have described the international context and the national history, which together form the background for a radically new and integral computing subject in Danish high schools. The new subject has been described in terms of two foundational theses, seven knowledge areas, and five didactical design principles.

6.1 Status

The status of the subject is that after the first year of the test period (2011-2012), 18% of the high schools taught the new subject. In the second (2012-2013, current) year of the test period, at least 26% of the high schools are teaching the new subject. Although no formal quantitative evaluation has yet been conducted, the informal feedback from teachers, examiners and pupils has been very positive.

As mentioned, the Danish Association of High School Teachers in Computing offers a number of learning activity packages on their website. Teachers are encouraged to develop and share their own learning activity packages. This bottom-up approach to material development of course encourage diversity and multiplicity, which challenges the content structure framework, and the conceptual framework, understanding and application of knowledge areas.

To reinforce the common understanding of the knowledge areas, a number of short reports have been developed by academics from Danish universities. Furthermore teacher training has been initiated in an ad-hoc fashion, offering 3 days of seminars during the winter of 2012, and again in the fall of 2012, where teachers are instructed in the use of the learning activity packages. Teachers from roughly 20% of all high schools attend these courses. While these ad-hoc seminars are necessary means in the process of developing the new subject, they are far from sufficient for fulfilling the requirements for in-service training of teachers to qualify them for teaching the new subject.

6.2 Plans

The plans for the continued development of the subject are fourfold:

- To further develop materials and resources
- To develop formal teacher training
- To establish professional learning communities
- To initiate relevant research
- To gain political interest and momentum

With respect to materials, we want to further iterate, increment and refine the content structure framework and the associated learning materials (both the knowledge area reports and the learning activity packages). A possi-

² <http://www.iftek.dk>

ble next step could be to develop free online materials supporting inversion of the classroom.

With respect to teacher training, we need to replace the current ad-hoc approach with a regular 120 ECTS education in computational thinking and practice to be offered to high school teachers – both pre-service and in-service.

We would like to support the former initiatives by further evolving the current formal and informal networks among high school computing teachers into professional learning communities based on action learning.

The new Danish initiative is an excellent opportunity for (and it deserves) a thorough treatment in terms of a number of related research projects. For example, we would like to investigate the following research questions:

- Why is computational thinking and computing practice generally and universally important to society and the individual?
- What are the relevant didactical design principles for the new subject?
- What is the ideal selection of knowledge areas for the new subject, and how do they compare to similar efforts internationally?
- How can we develop methodological and technological support for developing motivating and efficient learning activities that properly exploits the chosen didactical design principles?
- How can we develop efficient teacher training for the new subject?

Finally we find it of utmost importance, that we obtain political awareness about the importance of the subject, as it should not be an elective, but an integral, mandatory part of any high school education. A possible next step in this direction could be to host a conference on the importance of the subject.

6.3 Acknowledgments

This work was supported in part by Central Denmark Region and was conducted as part of the project Create IT which includes the partners: it-vest – networking universities, the Danish High School Computing Teachers Association (IFTEK), Egaa Gymnasium, and Centre for Science Education at Aarhus University.

The authors would like to thank Elisabeth Husum, Jakob Stenlökke Bendtsen, Bartłomiej Rohard Warszawski, Henning Agesen, and Peter Nørgaard for contributions and inspiring discussions. We would also like to thank the anonymous reviewers for constructive and relevant feedback.

7 References

- Agesen, H. and Nørgaard, P. (2009): *Investigation of Computing Subjects in High School* (in Danish: *Undersøgelse af IT fagudbuddet I de gymnasiale uddannelser*), Department for High Schools, Ministry of Education, Denmark.
- Andersen, P.B., Bennedsen, J., Brandorff, S., Caspersen, M.E., and Mosegaard, J. (2003): Teaching Programming to Liberal Arts Students — A Narrative Media Approach. *Proc. of the Conference on Innovation and Technology in Computer Science Education*, Thessalonica, Greece, **8**:109-113, ACM Press.
- Astrachan, O., Cuny, J., Stephenson, C., and Wilson, C. (2011): The CS10K Project: Mobilizing the Community to Transform High School Computing. *Proc. of the 42nd ACM Technical Symposium on Computer Science Education*, Dallas, TX, USA, **42**:85-86, ACM Press.
- Astrachan, O., Briggs, A., Cuny, J., Diaz, L., and Stephenson, C. (2012): Update on the CS Principles Project. *Proc. of the 43rd ACM Technical Symposium on Computer Science Education*, Raleigh, NC, USA, **43**:477-478, ACM Press.
- Atkinson, R.K., Derry, S.J., Renkl, A., and Wortham, D. (2000): Learning from Examples: Instructional Principles from the Worked Examples Research, *Review of Educational Research*, **70**(2):181-214.
- Bargury, I.Z. (2012): A New Curriculum for Junior-High in Computer Science. *Proc. of the Conference on Innovation and Technology in Computer Science Education*, Haifa, Israel, **17**:204-208, ACM Press.
- Bell, T., Andrae, P., & Lambert, L. (2010). Computer Science in New Zealand High Schools. presented at the meeting of the Twelfth Australasian Computing Education Conference (ACE 2010), Brisbane, Australia.
- Bell, T., Andrae, P., & Robins, A. (2012). Computer science in NZ high schools: the first year of the new standards. presented at the meeting of the 43rd ACM technical symposium on Computer Science Education, Raleigh, North Carolina, USA.
- Bennedsen, J. and Caspersen, M.E. (2004): Teaching Object-Oriented Programming – Towards Teaching a Systematic Programming Process. *Proc. of the Eighth Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts*, 18th European Conference on Object-Oriented Programming (ECOOP 2004), Oslo, Norway.
- Bennedsen, J., Caspersen, M.E., and Kölling, M. (2008): *Reflections on the Teaching of Programming*, Lecture Notes in Computer Science, Vol. 4821, Springer-Verlag.
- Caspersen, M.E. (2007): *Educating Novices in the Skills of Programming*, DAIMI PhD Dissertation PD-07-04, ISSN 1602-0448 (paper), 1602-0456 (online).
- Caspersen, M.E. and Bennedsen, J. (2007): Instructional Design of a Programming Course: A Learning Theoretic Approach. *Proc. of the International Computing Education Research Workshop*, Atlanta, Georgia, USA, **3**:111-122, ACM Press.
- Caspersen, M.E. and Kölling, M. (2009): STREAM: A First Programming Process, *ACM Transactions on Computing Education*, **9**(1):4.1-4.29.
- Christensen, H.B. and Caspersen, M.E. (2002): Frameworks in CS1: a Different Way of Introducing Event-Driven Programming. *Proc. of the Conference on Innovation and Technology in Computer Science Education*. Aarhus, Denmark, **7**:75-79.
- Chi, M.T.H., Bassok, M., Lewis, M.W., Reimann, P., and Glaser, R. (1989): Self-explanations: How students

- study and use examples in learning to solve problems, *Cognitive Science*, **13**(2):145-182.
- Cutts, Q., Esper, S., and Simon, B. (2011): Computing as the 4th “R”. *Proc. of the International Computing Education Research Workshop*, Providence, RI, USA, 7:133-138, ACM Press.
- Danish Ministry of Education (2011): Information Technology B and C, *Eight Reports With Guidelines for Information Technology B and C at stx, hf, htx, and hhx*, Department of High Schools, Ministry of Education. <http://www.uvm.dk/Uddannelser-og-dagtilbud/Gymnasiale-uddannelser/Studieretninger-og-fag/Forsøgsfag-i-de-gymnasiale-uddannelser/Informationsteknologi-C-og-B> (in Danish, accessed 24th August 2012).
- Denning, P. J. (2003): Great principles of computing. *Communications of the ACM*, **46**(11):15-20.
- Gal-Ezer, J. and Harel, D. (1998): What (Else) Should CS Educators Know?, *Communications of the ACM*, **41**(9):77-84.
- Gal-Ezer, J. and Harel, D. (1999): Curriculum and Course Syllabi for a High-School Program in Computer Science, *Computer Science Education*, **9**(2):114-147.
- Guzdial, M. (2008): Paving the Way for Computational Thinking, *Communications of the ACM*, **51**(8):25-27.
- Hadjerrouit, S. (2009): Teaching and Learning School Informatics: A Concept-Based Pedagogical Approach, *Informatics in Education*, **8**(2):227-250.
- Hazzan, O., Lapidot, T., and Ragonis, N. (2011): *Guide to Teaching Computer Science: An Activity-Based Approach*, Springer-Verlag.
- Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003): The BlueJ system and its pedagogy, *Computer Science Education*, **13**(4):249-268.
- LeFevre, J.-A. and Dixon, P. (1986): Do Written Instruction Need Examples?, *Cognition and Instruction*, **3**(1):1-30.
- Pattis, R.E. (1990): A philosophy and example of CS-1 programming projects. *Proc. of the 21st ACM Technical Symposium on Computer Science Education*, Washington D.C., USA, **21**:34-39, ACM Press.
- Perlis, A. (1962): The computer in the university. In *Computers and the World of the Future*, 180-219. Greenberger, M. (ed.). MIT Press.
- Royal Society (2012): *Shut down or restart? The way forward for computing in UK schools*. The Royal Society, UK.
- Rushkoff, D. (2010): *Program or Be Programmed – Ten Commands for a Digital Age*. New York, OR Books.
- Schmolitzky, A. (2005): Towards Complexity Levels of Object Systems Used in Software Engineering Education. *Proc. of the Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts*, 19th European Conference on Object-Oriented Programming (ECOOP 2005). Glasgow, UK.
- Steer, C. and Hubwieser, P. (2010): Comparing the Efficiency of Different Approaches to Teach Informatics at Secondary Schools, *Informatics in Education*, **9**(2):239-247.
- Van Diepen, N., Perrenet, J., and Zwaneveld, B. (2011): Which Way with Informatics in High Schools in the Netherlands? The Dutch Dilemma, *Informatics in Education*, **10**(1):123-148.
- Wing, J. (2006): Computational Thinking, *Communications of the ACM*, **49**(3):33-35.

How difficult are exams? A framework for assessing the complexity of introductory programming exams

Judy Sheard

Monash University
judy.sheard@monash.edu.au

Simon

University of Newcastle
simon@newcastle.edu.au

Angela Carbone

Monash University
angela.carbone@monash.edu.au

Donald Chinn

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

Tony Clear

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

Malcolm Corney

Queensland University of Technology
m.corney@qut.edu.au

Daryl D'Souza

MIT University
daryl.dsouza@rmit.edu.au

Joel Fenwick

University of Queensland
joelfenwick@uq.edu.au

James Harland

RMIT University
james.harland@rmit.edu.au

Mikko-Jussi Laakso

University of Turku
milaak@utu.fi

Donna Teague

Queensland University of Technology
d.teague@qut.edu.au

Abstract

Student performance on examinations is influenced by the level of difficulty of the questions. It seems reasonable to propose therefore that assessment of the difficulty of exam questions could be used to gauge the level of skills and knowledge expected at the end of a course. This paper reports the results of a study investigating the difficulty of exam questions using a subjective assessment of difficulty and a purpose-built exam question complexity classification scheme. The scheme, devised for exams in introductory programming courses, assesses the complexity of each question using six measures: external domain references, explicitness, linguistic complexity, conceptual complexity, length of code involved in the question and/or answer, and intellectual complexity (Bloom level). We apply the scheme to 20 introductory programming exam papers from five countries, and find substantial variation across the exams for all measures. Most exams include a mix of questions of low, medium, and high difficulty, although seven of the 20 have no questions of high difficulty. All of the complexity measures correlate with assessment of difficulty, indicating that the difficulty of an exam question relates to each of these more specific measures. We discuss the implications of these findings for the development of measures to assess learning standards in programming courses.

Keywords: Standards, quality, examination papers, CS1, introductory programming, assessment, question complexity, question difficulty.

1 Introduction

In Australia there has been an increasing amount of attention placed on the government's higher education standards agenda, which aims to achieve quality assurance in a number of areas including the standard of qualifications and the learning outcomes of students in higher education institutions. The Tertiary Education Quality and Standards Agency (TEQSA) has been established to register and evaluate the performance of higher education providers against a new Higher Education Standards Framework (Tertiary Education Quality and Standards Agency, 2012). To ensure that standards are developed the government has formed a Standards panel (Evans, 2011) to set the benchmark for quality in higher education.

The interest in learning standards is not restricted to government agencies. In a recent online survey of Australian academics, with more than 5,000 respondents across 20 universities, 46.7% of respondents felt that academic standards were in decline (Bexley et al, 2011). From the student perspective, in a survey of nearly 10,000 graduates in 2008, 67% nominated "Challenge students to achieve high academic standards" as an area of potential improvement for undergraduate education (Coates & Edwards, 2008).

In this environment, the challenge currently facing academics in the Australian tertiary sector is how to develop learning standards and assess learning outcomes. As a way forward, the Australian government has funded eight groups to work within specific disciplines to develop learning standards: the minimum required knowledge, skill and capabilities expected of a graduate. The combined discipline group for Information and Communication Technology (ICT) and Engineering has begun its quest for learning standards by drawing on existing learning outcomes developed from the relevant professional bodies (Cameron & Hadgraft, 2010).

Proposals currently under consideration to assess the attainment of learning standards include the development

of national standardised tests of generic and disciplinary learning outcomes. In the engineering field, a feasibility study is looking into testing discipline-specific skills as part of an Assessment of Higher Education Learning Outcomes (AHELO) (Australian Council for Educational Research, 2011). As yet there is no comparable venture in ICT, where the idea of standardised testing seems to be a difficult one to address. There appear to be no clear processes, pathways, or in some cases, communities, to offer a coherent way forward to assessment of discipline-specific learning standards.

The work reported here forms part of the BABELnot project (Lister et al, 2012), a principal goal of which is to explore a possible approach for the development and assessment of learning standards in programming courses. Formal written examinations are a common form of assessment in programming courses, and typically the form to which most marks are attached. The approach we have taken is to analyse examination papers to investigate levels of, and variations in, assessment of learning outcomes across institutions. In prior work (Simon, Sheard, Carbone, Chinn, et al, 2012) we analysed programming exam papers to identify the range of topics covered and the skills and knowledge required to answer exam questions in introductory programming. Here we extend that approach to determine the complexity and difficulty of exam questions and the level of knowledge required to answer them correctly. Our approach is similar to that taken by Crisp et al (2012), who explored the types of assessment tasks used to assess graduate attributes.

In this study we analyse exam questions to determine the levels of difficulty and complexity of the exams, which leads to an understanding of the standards being assessed. We first explore the concepts of task complexity and difficulty. We then develop a framework to measure the complexity of programming exam questions from which we can infer the level of achievement we expect from our programming students. Next, we apply this to a set of programming exam papers from multiple institutions to compare the levels of knowledge and skills being assessed.

2 Task Complexity and Task Difficulty

In a comprehensive review of the literature on the concept of task complexity, Campbell (1988) proposed that task complexity can be defined objectively as a function of task attributes that place high cognitive demands on the performer of the task. Braarud (2001) further distinguished between the *objective* task complexity, which is a characteristic of the task itself, and *subjective* task complexity, which is the user's perception of the complexity of a task. Both Campbell and Braarud argued that task *difficulty* is distinct from task complexity, incorporating additional aspects – such as the task context – that can entail high effort in doing a task. Campbell (1988) proposed that complex tasks are often ill-structured and ambiguous. He observed that while complex tasks are necessarily difficult, difficult tasks are not necessarily complex. For example, tracing a path through a maze with a pencil can be quite complex, but is seldom difficult.

Complexity is clearly a key concept for determining the difficulty of a task, but there seems to be little consensus amongst researchers about what attributes can be used to determine the complexity of a task. Campbell (1988) proposed four properties that influence task complexity and used these to develop a task typology. Mennin (2007) distinguished between simple, complicated and complex problems, but did not explain the distinction, instead using examples to illustrate the categories. Haerem and Rau (2007) developed an instrument to measure variability and analysability, which they suggested are fundamental aspects of complexity. An investigation of task complexity by Stahl, Pieschl and Bromme (2006) used Bloom's taxonomy to classify tasks of different levels of complexity. They further classified according to level of difficulty within these tasks, but did not define what they meant by this.

Williams and Clarke (1997) completed the most comprehensive work in this area. They proposed six dimensions of complexity (linguistic, contextual, representational, operational, conceptual and intellectual) and applied these to problems in the mathematics domain. Carbone (2007) later applied these six dimensions to tasks in the computer programming domain.

3 Exam Question Complexity

In our work we wished to investigate the level of difficulty of exam questions as a means to assess the level of skills and knowledge being tested in introductory programming courses. A search of the literature on programming exam questions indicated a number of factors that contribute to the complexity and hence difficulty of these assessment tasks.

A common factor identified was the *cognitive load* placed on the student by the question, which is defined as the number of discrete pieces of information that the student is required to understand in order to answer the question (Sweller, 1988). An investigation of second-year data structures exam questions by Simon et al (2010) proposed that the phrasing and construction of a question can add to cognitive load and therefore increase the difficulty of a question. They argued that cognitive load also increases when questions involve multiple concepts. In a review of 15 introductory programming exams from 14 schools, Petersen et al (2011) investigated the content and concepts covered by each question, proposing that the more concepts the students need to deal with to answer a question, the higher the cognitive load and hence the difficulty of the question. They assessed cognitive load simply by counting the distinct concepts dealt with in a question, without considering whether different concepts might have different intrinsic levels of difficulty. They found that code-writing questions had the highest number of concepts per question. In a study of data structures exams, Morrison et al (2011) found few long questions, and proposed that this was due to the exam setters wishing to avoid the increased cognitive load that would come with extra length.

The conceptual level of topics covered by a question has also been proposed as an influence on question difficulty. A survey by Schulte and Bennedsen (2006) gathered 242 academics' opinions of the difficulty of CS1 topics. The topics found most difficult were design,

Measure	Focus*	Classification values	Description
External domain references	Q only	low, medium, high; if medium or high, the external domain is specified	Reference to a domain beyond what one would reasonably expect introductory programming students to know
Explicitness	Q only	high, medium, low (note order of levels)	Extent of prescriptiveness as to how to answer the question
Linguistic complexity	Q only	low, medium, high	Length and sophistication of the natural language used to specify the question
Conceptual complexity	Q & A	low, medium, high	Classification of the individual programming concepts required to answer the question
Code length	Q & A	low, medium, high, NA	Whether code is up to half a dozen lines long, up to two dozen lines long, or longer
Intellectual complexity (Bloom level)	Q & A	knowledge, comprehension, application, analysis, evaluation, synthesis	Bloom's taxonomy as applied to programming questions by Gluga et al (2012)
Level of difficulty	Q & A	low, medium, high	Subjective assessment of difficulty of question

* The second column, Focus, indicates whether the measures apply only to the question or to the question and answer.

Table 1: Six complexity measures and level of difficulty used to classify exam questions

recursion, advanced OO topics (polymorphism & inheritance) and pointers & references. This aligns well with a survey of 35 academics by Dale (2006) which showed that design, problem solving, control structures, I/O, parameters, recursion, and OO concepts were seen as the difficult concepts for novice programming students.

Based on a detailed statistical analysis of student answers to introductory programming exam questions, Lopez et al (2008) proposed a hierarchy of programming-related skills. In an attempt to interpret results that were not intuitively obvious they concluded that there were characteristics of a task other than its style that could explain its level of difficulty. They proposed that the size of the task and the programming constructs used also influenced the difficulty of a question.

A corpus of work has used Bloom's taxonomy (Anderson & Sosniak, 1994) to classify questions according to the cognitive demand of answering them (Thompson et al, 2008); or the SOLO taxonomy (Hattie & Purdie, 1998) to classify the intellectual level demonstrated by answers to questions (Clear et al, 2008; Sheard et al, 2008).

From a different perspective, the study of engineering exam questions by Goldfinch et al (2008) concluded that the style and structure of questions influenced perceptions of difficulty.

4 Classifying Exam Question Complexity

In the studies of programming exam questions that we have reviewed, the assessments of question difficulty were impressionistic; however, the reasons given for difficulty usually pointed to specific aspects of complexity. Some of these related to the question itself, some to what was required as a response. We considered that complexity could be inherent both in the question and in the response to the question. This led us to propose a framework for assessing the aspects of complexity of exam questions which could then be used to identify areas of difficulty for the student.

To determine the factors that influence complexity we considered four perspectives.

1. How is the question asked? How readily will the students be able to understand what the question is asking them to do? Question phrasing or style can lead to ambiguity and uncertainty in how to respond

(Goldfinch et al, 2008; Simon et al, 2010). To address these questions we consider the *linguistic* complexity of the question and references to *external domains* beyond the scope of the course, which we called 'cultural references' in our previous work (Sheard et al, 2011).

2. How much guidance does the question give as to how it should be answered (Goldfinch et al, 2008; Simon et al, 2010)? Here we consider the *explicitness* of the question.
3. What is the student required to do in order to answer the question? Here we consider the *amount of code* to be read and/or written and the *intellectual complexity* level demanded (Lopez et al, 2008; Morrison et al, 2011; Petersen et al, 2011).
4. What does the student need to understand in order to answer the question? This relates to the number of concepts involved in the question and to their intrinsic complexity. Here we consider the *conceptual* complexity (Dale, 2006; Schulte & Bennedsen, 2006).

The aspects of complexity highlighted by these questions led to the development of an exam question complexity classification scheme, a framework for determining the levels of complexity of a question. There are six dimensions to the scheme, as shown in the first six rows of Table 1. The first three are concerned with the exam question alone and the next three are concerned with the question and answer combined. For example, linguistic complexity applies to the language in which the question is expressed, while code length assesses the combined length of any code provided in the question and any code that the student is required to write in the answer. Four of the six measures of complexity are closely aligned to the dimensions of Williams and Clarke (1997). These are external domain references and linguistic, conceptual and intellectual complexities. The last row of the table shows the measure of level of difficulty, which we consider to be distinct from the various measures of complexity.

For each measure, the possible classification values and a brief description of the measure are given. More detailed explanations of the complexity measures are given in the results section.

5 Research Approach

The first version of the exam question complexity classification scheme emerged from a brainstorming session that was framed by the perspectives listed in Section 4 and informed by the literature discussed in the preceding sections. This was followed by a number of iterations in which about a dozen researchers applied the measures to the questions in an exam. After each round of classification the measures were clarified and adjusted as appropriate until the classification scheme appeared to have stabilised.

At that point an inter-rater reliability test was conducted on the complexity measures, with all researchers classifying all 37 questions in a single exam, first individually and then in pairs. As all of the measures are ordinal, reliability was calculated using the Intraclass Correlation (ICC) (Banerjee et al, 1999), and was found to be satisfactory on all measures, with pairs proving distinctly more reliable than individuals.

Following this test, the remaining exams were classified by pairs of researchers, who first classified the questions individually and then discussed their classifications and reconciled any differences.

6 Results

This section presents the results of analysing 20 introductory programming exam papers using the complexity measures listed in Table 1. A total of 472 questions were identified in these exams, with the number of questions in an exam ranging from four to 41.

The 20 exam papers were sourced from ten institutions in five countries. All were used in introductory programming courses, 18 at the undergraduate level and two at the postgraduate level. The latter two courses are effectively the same as courses taught to first-year undergraduate students, but are taught to students who are taking a postgraduate computing qualification to supplement a degree in some unrelated area. Course demographics varied from 25 students on a single campus to 800 students over six campuses, two of these being overseas campuses. Most courses used Java with a variety of IDEs (BlueJ, JCreator, Netbeans, Eclipse), two used JavaScript, one used C# with Visual Studio, one used Visual Basic, one used VBA (Visual Basic for Applications), and one used Python.

Most of the exams were entirely written, but two were separated into a written part and a computer-based part, each worth 50% of the complete exam.

Note that for the analysis, the percentage mark allocated to each question has been used as a weighting factor for the other measures.

6.1 Overall Complexity Measures

Each question was classified according to six measures of complexity.

The results for five levels of complexity (external domain references, explicitness, linguistic complexity, conceptual complexity and code length) are summarised in Table 2. Because the percentage mark allocated to each

Measure of complexity	low	medium	high
External domain references	95%	5%	0%
Explicitness	3%	30%	67%
Linguistic complexity	80%	17%	3%
Conceptual complexity	8%	67%	25%
Code length*	27%	54%	10%

* 9% of questions (weighted) did not involve code

Table 2: Overall levels of complexity of questions from the 20 exams, with mode values shown in bold

question was used as a weighting, the figures in the table represent the percentage of the exam marks allocated, not the percentage of the number of questions.

While these five measures are all classified as low, medium, or high, intellectual complexity is classified according to Bloom's six-point scale, so its classifications are shown separately in Figure 1. These classifications ranged from 3% for Evaluation to 44% for Application.

Considering the mode values, we can see that, over all the exams, questions are predominantly low in external domain references, highly explicit, low in linguistic complexity, of medium conceptual complexity and medium code length, and at the Application level of Bloom's taxonomy.

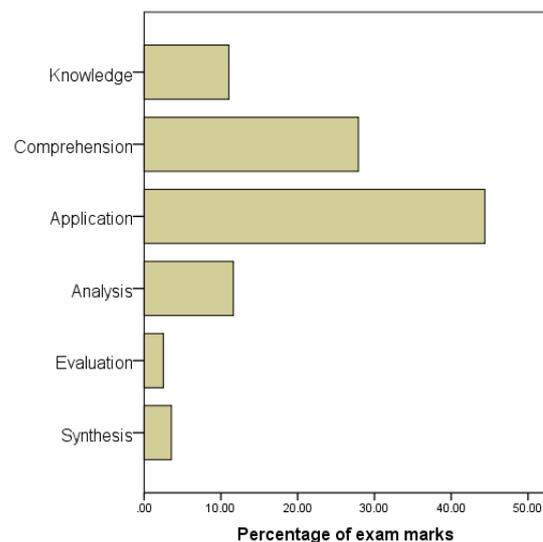


Figure 1: Overall measure of intellectual complexity (Bloom's taxonomy)

6.2 External Domain References

Many exam questions involve some sort of scenario, referring to a domain beyond what would necessarily be taught in the programming course. These scenarios have the potential to make a question more complex.

An external domain reference is any use of terms, activities, or scenarios that may be specific to a particular group and may reduce the ability of those outside the group to understand the question. For example, if a question refers to the scoring scheme of Australian Rules football, students would require specific knowledge to fully understand it – unless the question explicitly includes all of the knowledge that is needed to deduce the answer. Another question might display a partly complete backgammon game and ask students to write a program to determine the probability of winning on the next throw

of the dice. Unless the question fully explains the relevant rules, students who do not know backgammon will clearly not be able to answer it.

Programming knowledge does not constitute an external domain reference, because it is assumed to be taught in the course or prior courses. General knowledge is not considered as an external domain reference so long as the classifier is confident that it really is general: that all introductory programming students could reasonably be expected to know it.

We classify external domain references as high if students cannot understand the question without knowing more about an external domain; medium if they are given all the information they need, but the wording might lead them to think otherwise; and low if all students should be able to understand the question as it is.

None of the questions that we analysed relied upon a high level of knowledge from an external domain. Only seven exams contained questions with a medium level of external domain knowledge; these comprised at most 20% of any exam, and made up only 23 questions (less than 1% of the 472 questions).

Of those 23 questions, a few assumed some knowledge of the business domain (interest, profit, taxes), and a couple assumed knowledge of mathematical concepts (complex numbers, log arithmetic) or scientific concepts (storm strength). A few questions assumed knowledge of computing concepts (codes/encryption, pixellation, domain name format) beyond what would be considered reasonable for an introductory programming student. Some references were culturally based (name format, motel, vehicle registration, sports, card games). One referred to a sorting hat, a concept from a popular series of books and films. All of these references were at the medium level, so students did not need the external domain knowledge in order to answer the questions.

Of particular interest are questions that refer to external domain knowledge but make it clear that this knowledge was covered thoroughly during the course, perhaps being the subject of a major assignment. Such questions would not constitute external domain knowledge for this particular cohort of students. However, if the question were to be placed in a repository for the use of other academics, it would be wise to flag that there are external domain references for other students; therefore such references were classified as requiring external domain knowledge, with the domain in question being specified as the course assignment.

6.3 Explicitness

How strongly does the question tell the student what steps to use in writing an answer? How strongly does it prescribe, for example, what programming constructs and/or data structures to use? There is a fairly high level of explicitness in “Write a method that takes an array of integers as a parameter and returns the sum of the numbers in the array”. There is a very low level of explicitness in “Write a program to simulate an automatic vending machine,” which requires the students to determine the purchase process of the vending machine and identify the corresponding programming operations. Another question might require students to specify a Card class to use in a card game program. A highly explicit

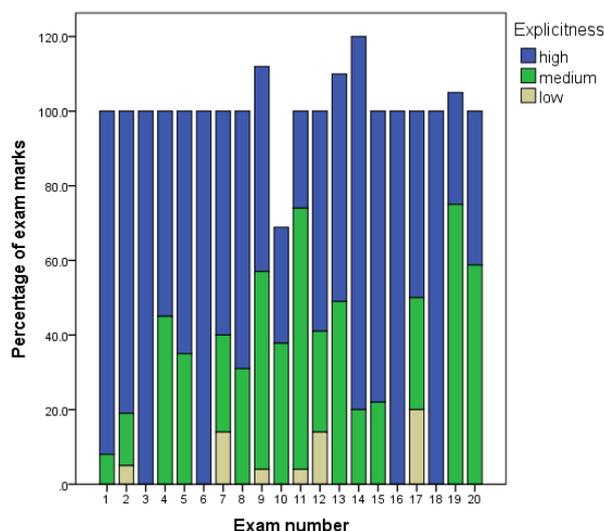


Figure 2: Explicitness of questions in each exam

version would list the methods required and the attributes and their types. A version with a low level of explicitness would not specify methods, attributes, or operations. A version with medium explicitness would perhaps specify some of the attributes and/or methods and require the student to deduce the rest. Note that the level of explicitness of a question would be expected to have an inverse relationship with the question’s difficulty; that is, the more explicit a question, the easier we would expect students to find it.

Figure 2 shows a summary of the explicitness levels of questions over the 20 exams. Two thirds of the marks (67%) were allocated to questions expressed with a high level of explicitness. The graph shows that less than a third of the exams (6) contained questions of low explicitness, and the marks allocated for these questions comprised 20% or less of these exams. In most of the exams more than half the questions were highly explicit.

Note: in Figure 2, and those following, 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 exam that is less than 100% is from a course that included non-programming topics and has questions that do not relate to programming. We classified only the programming-related questions in this exam.

6.4 Linguistic Complexity

Linguistic complexity is related to the length and sophistication of the natural language used to specify the question. Some questions have lengthy descriptions or use unusual words, which could affect the ability of a student to answer them. One possible view of linguistic complexity is that it is an approximation of the likelihood that a student not fluent in the natural language of the question would have trouble understanding the question.

Overall, most marks were allocated to questions involving a low level of linguistic complexity (80%). In about a third of the exams (7), all questions were classified as having a low level of linguistic complexity. Only two exams had questions with high linguistic complexity. In one of these, the high linguistic complexity was in a single question, comprising 50% of the exam, which was to be answered at the computer.

6.5 Conceptual Complexity

Questions in programming exams usually require students to understand a number of different ideas or concepts. On the basis of our own experience of teaching introductory programming, and of other people’s survey findings (Dale, 2006; Schulte & Bennedsen, 2006), we have classified a number of programming concepts as being of low, medium, or high conceptual complexity. For example, variables and arithmetic operators are of low conceptual complexity; methods, and events are of medium conceptual complexity; and recursion, file I/O, and arrays of objects are of high conceptual complexity. Note that these levels were defined specifically for Java-like procedural or object-oriented introductory programming courses; when we come to classify exams in courses that use functional programming, we might need to redefine them, with recursion, for example, shifting from high to a lower conceptual complexity.

We classified the questions using the initial categorisation as a guide, while remaining conscious that particular usage might affect the classification. For example, although loops are generally classified as medium, a classifier could argue for high complexity when classifying a particularly tortuous loop.

The conceptual complexity findings are summarised in Figure 3. Overall, two thirds of the marks (67%) were allocated to questions involving a medium level of conceptual complexity. Most exams showed a range of conceptual complexity, with the majority of marks allocated to questions involving concepts of medium complexity. Only four exams had no questions of high conceptual complexity, and only four had no questions of low conceptual complexity.

6.6 Code Length

The questions were classified according to the amount of code involved in reading and answering the question, with a simple guide that up to about half a dozen lines of code would be considered low, between there and about two dozen lines would be considered medium, and any more than about two dozen lines of code would be considered high. A summary of the results is shown in

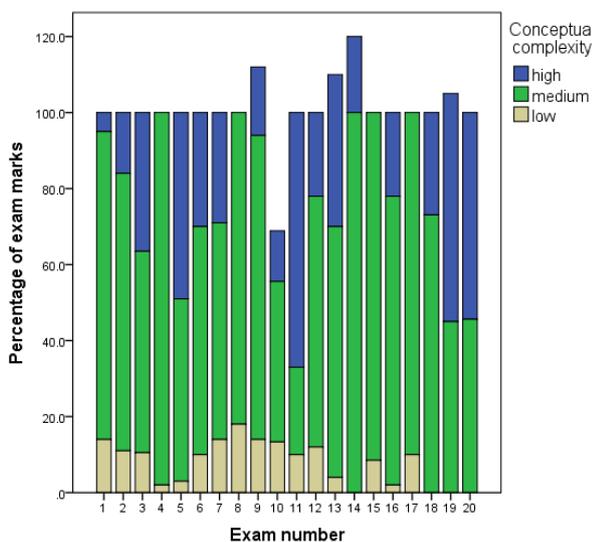


Figure 3: Conceptual complexity of questions in each exam

Figure 4. Overall, more than half the marks (54%) were allocated to questions involving a medium amount of code. About two thirds of the exams (14) contained questions that did not involve code; however, these were usually only a small component of the exam. Most of the marks in most of the exams were allocated to questions involving low and medium code length. Less than half the exams (8) had questions involving large amounts of code, and only three exams had large weightings of marks (more than 40%) involving high code length.

6.7 Intellectual Complexity

Bloom’s cognitive domain is a long-recognised measure of the intellectual complexity of a question in terms of its expected answer. There has been debate about whether Bloom’s domains can be usefully applied to programming questions, but there is some consensus that they can (Thompson et al, 2008). Gluga et al (2012) provide a clear explication, with examples and a tutorial, of one way of doing this.

The summary in Figure 5 shows a great variation in levels of intellectual complexity across the exams. Considering the three lowest levels of intellectual complexity, most exams (17) contained questions at the Knowledge level, all exams contained questions at the Comprehension level, and all but one exam contained questions at the Application level. Questions at the Analysis level were found in just over half the exams (12). At the highest Bloom levels there were very few questions, with only one exam containing Evaluation level questions and three exams containing Synthesis level questions.

6.8 Degree of Difficulty

Assessing the degree of difficulty entails classifying a question according to how difficult an average student at the end of an introductory programming course is likely to find it. This is a holistic measure. We would expect there to be a correlation between question difficulty and students’ marks on the question: the higher the difficulty, the lower the average mark we might expect students to attain.

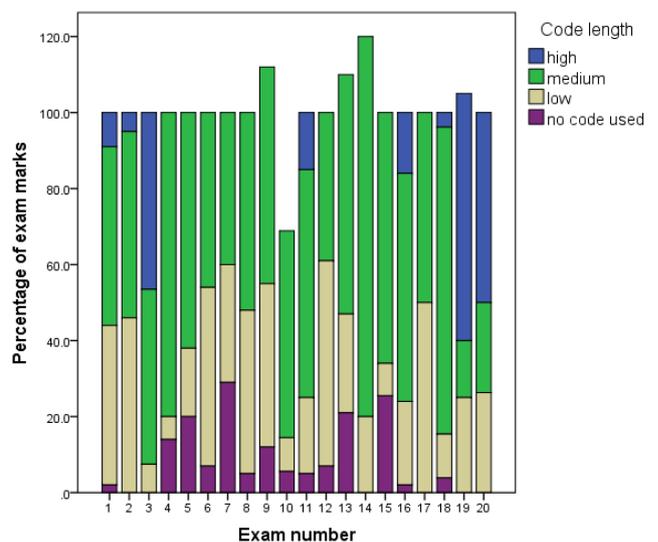


Figure 4: Length of code involved in questions in each exam

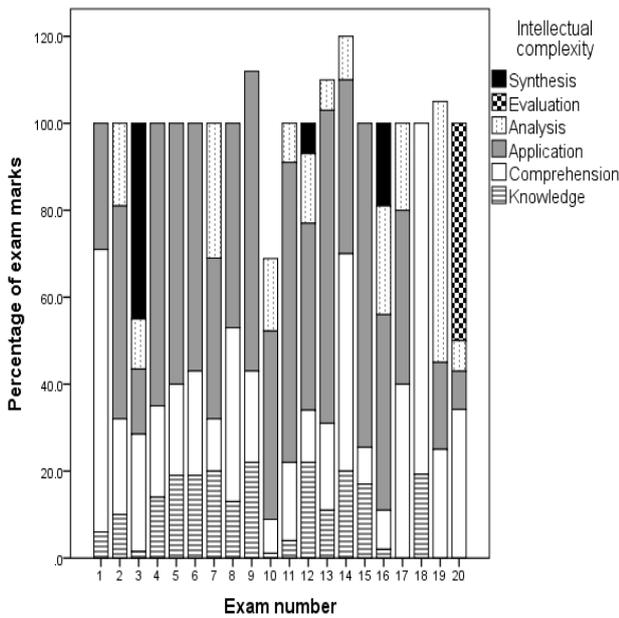


Figure 5: Intellectual complexity of questions in each exam

Question difficulty assesses the student’s ability to manage all of the demands of a task. It is concerned with the student’s perception of and response to the question, whereas task complexity is static and defined by the nature of the question itself.

The questions were classified according to the perceived level of difficulty for a student at the end of an introductory programming course. Overall, half the marks (50%) were allocated to questions rated as medium, with 30% for questions rated as low difficulty and 20% for question rated as high difficulty. The summary in Figure 6 shows the variation across the exams. Although some exams have a fairly wide spread of low/medium/high difficulty questions, about a third (7) of the exams have no high difficulty questions, and one exam has high difficulty only in a bonus question. One exam had no questions of low difficulty, just 45% medium and 55% high.

We have been asked why we bother to subjectively assess question difficulty when the students’ marks on the questions would provide a more reliable measure. The answer is simple. We were fortunate enough to have been provided with these 20 exams to analyse. It would be too much to have also asked for student performance data on each question of each exam. First, it is possible that for many of the exams the only data now available is students’ overall marks in the course, and perhaps even that is no longer available. Second, ethics approval is required before students’ results can be analysed for research purposes, and we did not feel it appropriate to ask everyone who gave us an exam to follow this up by applying for ethics approval in order to give us their students’ results as well.

However, we have conducted a separate study (Simon, Sheard, Carbone, D'Souza, et al, 2012), on a set of questions for which we do have access to student performance data, and have confirmed the expected link between our assessment of question difficulty and the students’ performance on the same questions.

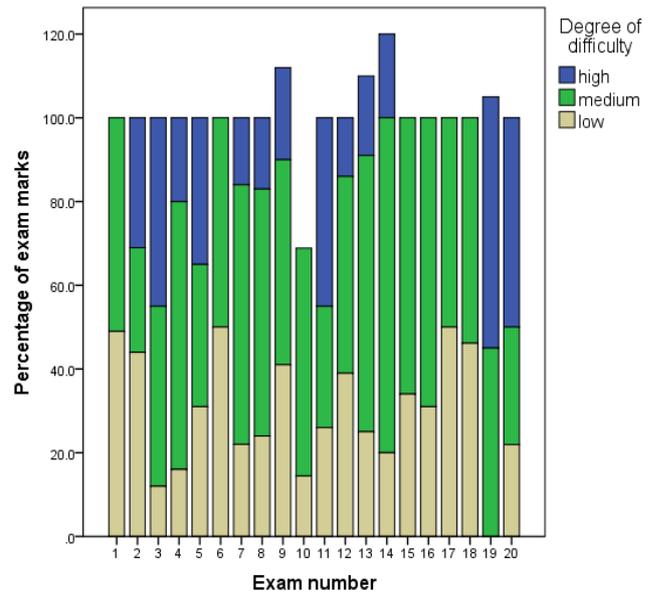


Figure 6: Degree of difficulty of questions in each exam

6.9 Relationship between Complexity and Difficulty

Each of the measures of complexity focuses on particular characteristics of a question which could be seen to contribute to an overall complexity for the question, whereas the degree of difficulty is a perception of how difficult the average student at the end of the introductory programming course would find the question. A correlation test was performed to explore the relationship between complexity and difficulty. As the measures of complexity and difficulty are at the ordinal level, a Spearman’s Rank order correlation was used. The results, summarised in Table 3, show relationships between the degree of difficulty and each measure of complexity, all of which are significant at $p < 0.01$. The strongest relationships with degree of difficulty are code length and intellectual complexity: questions involving more program code, and questions at the higher levels of Bloom’s taxonomy, are more difficult questions.

To further explore the relationship between complexity and difficulty, the levels of difficulty within each complexity measure were determined. The following results were found.

- Most questions with a low level of difficulty are highly explicit (97%).
- Most questions with a low level of difficulty are of low linguistic complexity (98%).

	Degree of difficulty (r)
External domain references	0.197 *
Explicitness	-0.408 *
Linguistic complexity	0.326 *
Conceptual complexity	0.412 *
Code length	0.564 *
Intellectual complexity	0.501 *

* all significant at $p < 0.01$

Table 3: Relationship between degree of difficulty and complexity measures

- No questions with a high level of difficulty have a low conceptual complexity.
- No questions with a high level of difficulty involve a low amount of code or no code.
- No questions with low level of difficulty involve a high amount of code.
- No questions with a high level of difficulty are classified at the two lowest levels of Bloom's cognitive domain (Knowledge and Comprehension). Figure 7 shows the breakdown of the degree of difficulty within each level of Bloom's taxonomy.

7 Discussion

We have found from our analysis that there is wide variation in the final examinations of introductory programming courses, with variations in all the complexity measures and in the level of difficulty. As the pressure grows to determine standards of assessment for university courses, the framework that we have devised is likely to prove extremely helpful. We do not propose that all introductory programming courses should be identical, or that they should all assess at the same level; what we do propose is that there should be a means to determine the extent of similarity between the courses and their outcomes, a means to compare the levels at which they assess their students. As we see it, the push to standardise is not an attempt to impose uniformity but a desire to be explicitly aware of the spread and variety of what is taught and assessed.

An interesting consequence of our findings is that, notwithstanding their substantial overlap, different introductory programming courses do assess somewhat different material at somewhat different levels. Students migrating between programs, and academics charged with assigning credit on the basis of courses completed elsewhere, would do well to be aware of this.

Of the complexity measures addressed in this work, it is useful to distinguish between 'good' complexity and 'bad' complexity. High-level external domain references

and high linguistic complexity can be undesirable, and we were pleased to see little evidence of those in the exams we assessed. Conceptual and intellectual complexity can be intentional and purposeful, and it seems quite reasonable to test these to some extent – through there is still an open question as to what levels we can reasonably expect students to attain in an introductory programming course.

With regard to intellectual complexity, academics from other disciplines might be surprised to see such a preponderance of questions at the Application level in the exam for an introductory course. In other disciplines it might be expected that the first course will deal more with Knowledge and Comprehension, with the higher levels of the taxonomy reserved for higher-level courses. If this is indeed the case, we need to be confident that this high level of Application is a necessary consequence of the nature of teaching programming; the alternative is to recognise that we are asking too much sophistication of students in our introductory courses.

Does it help students or hinder them to have a practical, computer-based exam? Is it more acceptable in a computer-based exam than a paper-based exam to have a large question, worth 50% of the exam, that has high linguistic complexity, high conceptual complexity, high code length, a Bloom level of Evaluation, and a high perceived overall difficulty? We do not propose answers to these questions. Rather, we note that they have emerged from our study of these exams, and that they are worthy of consideration by the computing education community.

The assignment of topics to low, medium, and high conceptual complexity, while certainly not arbitrary, is clearly open to debate. The choices appear to have been reasonable, given the correlation between this measure and the overall question difficulty. However, we need to consider whether conceptual complexity is an intrinsic feature of a topic, or more a function of what was taught and how it was taught in each course. Just as the conceptual complexity of recursion might be high in a procedural programming course and low in a functional programming course, might it be the case that the conceptual complexity of any topic is dependent on when and how that topic was taught in each specific course? We also note in passing that while selection and iteration appear as topics in the surveys on which our own lists of topics were based, the topic of sequence is notable by its absence, although it has been identified by Simon (2011) and others as a topic that some students have difficulty grasping. In retrospect, we accept that it would have been wise to list sequence as a topic, assigning it a low level of conceptual complexity.

With regard to the various complexity measures described in this paper, is it possible and reasonable to suggest what mix of low, medium, and high values should normally be found in an introductory programming exam? Can we use these measures to suggest that particular exams are inappropriately complex or inappropriately simple? Or do we accept that there is a wide variety in the courses themselves, and simply note where each exam fits into the broader picture?

For some of the measures it is possible to make clear recommendations to the people who write exams. It

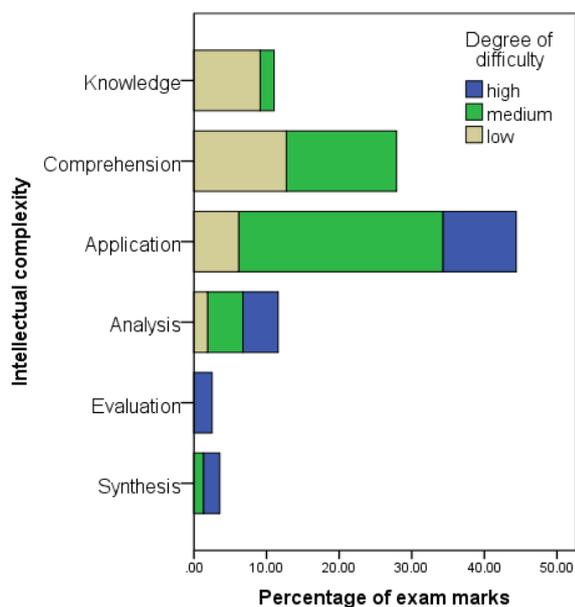


Figure 7: Degree of difficulty of questions within each level of intellectual complexity

would appear reasonable to expect exam questions to have low linguistic complexity and not to rely on students' knowledge of domains outside what is being taught. For other measures the choice is more personal. For example, some examiners might like to be entirely explicit about what students are required to do, while others might prefer to test the students' problem-solving abilities by framing some questions with low explicitness and leaving the students to fill in the gaps in the specifications. However, in a couple of exams we assessed, more than 75% of the questions were of medium level explicitness; would most examiners consider this a little high for an introductory programming exam?

The analysis reported in this paper is exploratory: its purpose is as much to identify questions as to answer them. Its contribution is that it raises questions such as those discussed above, at the same time providing a framework in which the questions can be discussed, and possibly, eventually, answered.

8 Conclusions and Future Work

In this study we analysed programming examination papers across institutions, both national and international, as a window into the levels of learning expected in foundation programming courses. The complexity measures applied in this study highlight the variability of introductory programming exams. This could be taken as reinforcing the suggestion that exams are highly personal; but it leaves open the question: are the exams all assessing the same or comparable things? If not, can we be sure that each and every one of these exams is a valid assessment instrument?

Future work will include exploring the thinking of the people who write the exams, and whether they do so with any awareness of the sorts of issue addressed in this analysis. This will entail interviewing a number of exam writers and conducting a qualitative analysis of the interview transcripts.

In addition, we intend to analyse a number of introductory programming exams that use functional programming, and to extend our analysis to the exams for second- and third-level programming courses.

With regard to the increasing role of standardisation, further aspects of the BABELnot project (Lister et al, 2012) include the establishment of a repository of fully classified programming exam questions with accompanying performance data, and the benchmarking of a subset of these questions across multiple institutions.

9 Acknowledgements

The authors would like to thank the Learning and Teaching Academy of the Australian Council of Deans of ICT for its support of the ACE 2012 workshop, and also the Australian Federal Government's Office for Learning and Teaching for its support of the BABELnot project.

10 References

Anderson, L. W. and Sosniak, L., A. (1994): Excerpts from the "Taxonomy of Educational Objectives, The Classification of Educational Goals, Handbook I: Cognitive Domain. In L. W. Anderson & L. Sosniak, A. (Eds.), *Bloom's Taxonomy: A Forty Year*

- Retrospective* (pp. 9-27). Chicago, Illinois, USA: The University of Chicago Press.
- Australian Council for Educational Research. (2011): Assessment of Higher Education Learning Outcomes (AHELO) Retrieved 24 August, 2012, from <http://www.acer.edu.au/aheloau>
- Banerjee, M., Capozzoli, M., McSweeney, L. and Sinha, D. (1999): Beyond kappa: a review of interrater agreement measures. *Canadian Journal of Statistics*, **27**, 3-23.
- Bexley, E., James, R. and Arkoudis, S. (2011): The Australian academic profession in transition. Canberra: Department of Education, Employment and Workplace Relations, Commonwealth of Australia.
- Braarud, P. (2001): Subjective task complexity and subjective workload: Criterion validity for complex team tasks. *International Journal of Cognitive Ergonomics*, **5**(3), 261-273.
- Cameron, I. and Hadgraft, R. (2010): Engineering and ICT Learning and Teaching Academic Standards Statement. Strawberry Hills, NSW, Australia: Australian Learning and Teaching Council.
- Campbell, D. (1988): Task complexity: A review and analysis. *Academy of Management Review*, **13**(1), 40-52.
- Carbone, A. (2007): *Principles for Designing Programming Tasks: How task characteristics influence student learning of programming*. PhD, Monash University, Melbourne, Australia.
- Clear, T., Whalley, J., Lister, R., Carbone, A., Hu, M., Sheard, J., Simon, B. and Thompson, E. (2008): *Reliably classifying novice programmer exam response using the SOLO taxonomy*. Paper presented at the 21st Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2008), Auckland, New Zealand.
- Coates, H. and Edwards, D. (2008): The 2008 Graduate Pathways Survey. Canberra: Department of Education, Employment and Workplace Relations, Commonwealth of Australia.
- Crisp, G., Barrie, S., Hughes, C. and Bennison, A. (2012): *How can I tell if I am assessing learning outcomes appropriately?* Paper presented at the Higher Education Research and Development Society of Australasia (HERDSA), Macquarie Hotel, Hobart. <http://conference.herdsa.org.au/2012/>
- Dale, N. (2006): Most difficult topics in CS1: Results of an online survey of educators. *inroads - The SIGCSE Bulletin*, **38**(2), 49-53.
- Evans, C. (2011): Professor Alan Robson to take on key higher education quality role *Media Release* Retrieved 24 August, 2012, from <http://ministers.deewr.gov.au/evans/professor-alan-robson-take-key-higher-education-quality-role>
- Gluga, R., Kay, J., Lister, R., Kleitman, S. and Lever, T. (2012): *Coming to terms with Bloom: An online tutorial for teachers of programming fundamentals*. Paper presented at the 14th Australasian Computing Education conference, Melbourne, Australia.
- Goldfinch, T., Carew, A. L., Gardner, A., Henderson, A., McCarthy, T. and Thomas, G. (2008): *Cross-institutional comparison of mechanics examinations:*

- A guide for the curious*. Paper presented at the Australasian Association for Engineering Education conference (AAEE), Yeppoon.
- Haerem, T. and Rau, D. (2007): The influence of degree of expertise and objective task complexity on perceived task complexity and performance. *Journal of Applied Psychology*, **92**(5), 1320-1331.
- Hattie, J. and Purdie, N. (1998): The SOLO model: Addressing fundamental measurement issues. In M. Turpin (Ed.), *Teaching and Learning in Higher Education* (pp. 145-176). Camberwell, Victoria, Australia: ACER Press.
- Lister, R., Corney, M., Curran, J., D'Souza, D., Fidge, C., Gluga, R., Hamilton, M., Harland, J., Hogan, J., Kay, J., Murphy, T., Roggenkamp, M., Sheard, J., Simon and Teague, D. (2012): *Toward a shared understanding of competency in programming: An invitation to the BABELnot project*. Paper presented at the 14th Australasian Computing Education conference, Melbourne, Australia.
- Lopez, M., Whalley, J., Robbins, P. and Lister, R. (2008): *Relationships between reading, tracing and writing skills in introductory programming*. Paper presented at the Fourth International Computing Education Research workshop (ICER 2008), Sydney, Australia.
- Mennin, S. (2007): Small-group problem-based learning as a complex adaptive system. *Teaching and Teacher Education*, **23**, 303-313.
- Morrison, B., Clancy, M., McCartney, R., Richards, B. and Sanders, K. (2011): *Applying data structures in exams*. Paper presented at the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE'11), Dallas, Texas, USA.
- Petersen, A., Craig, M. and Zingaro, D. (2011): *Reviewing CSI exam question content*. Paper presented at the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE'11), Dallas, Texas, USA.
- Schulte, C. and Bennedsen, J. (2006): *What do teachers teach in introductory programming?* Paper presented at the Second International Computing Education Research workshop (ICER'06), Canterbury, UK.
- Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E. and Whalley, J. (2008): *Going SOLO to assess novice programmers*. Paper presented at the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'08), Madrid, Spain.
- Sheard, J., Simon, Carbone, A., Chinn, D., Laakso, M.-J., Clear, T., de Raadt, M., D'Souza, D., Harland, D., Lister, R., Philpott, A. and Warburton, G. (2011): *Exploring programming assessment instruments: a classification scheme for examination questions*. Paper presented at the Seventh International Computing Education Research workshop (ICER 2011), Providence, Rhode Island, USA.
- Simon (2011): *Assignment and sequence: why some students can't recognise a simple swap*. Paper presented at the 10th Koli Calling International Conference on Computing Education research, Finland.
- Simon, Sheard, J., Carbone, A., Chinn, D., Laakso, M.-J., Clear, T., de Raadt, M., D'Souza, D., Lister, R., Philpott, A., Skene, J. and Warburton, G. (2012): *Introductory programming: Examining the exams*. Paper presented at the 14th Australasian Computing Education conference, Melbourne, Australia.
- Simon, Sheard, J., Carbone, A., D'Souza, D., Harland, J. and Laakso, M.-J. (2012): *Can computing academics assess the difficulty of programming examination questions?* Paper presented at the 11th Koli Calling International Conference on Computing Education Research, Finland.
- Simon, B., Clancy, M., McCartney, R., Morrison, B., Richards, B. and Sanders, K. (2010): *Making sense of data structure exams*. Paper presented at the Sixth International Computing Education Research workshop (ICER 2010), Aarhus, Denmark.
- Stahl, E., Pieschl, S. and Bromme, R. (2006): Task complexity, epistemological beliefs and metacognitive calibration: An exploratory study. *Journal of Educational Computing Research*, **35**(4), 319-338.
- Sweller, J. (1988): Cognitive load during problem solving. Effects on learning. *Cognitive Science*, **12**(2), 257-285.
- Tertiary Education Quality and Standards Agency. (2012): Higher Education Standards Framework, from <http://www.teqsa.gov.au/higher-education-standards-framework>
- Thompson, E., Luxton-Reilly, A., Whalley, J., Hu, M. and Robbins, P. (2008): *Bloom's taxonomy for CS assessment*. Paper presented at the 10th Australasian Computing Education Conference (ACE 2008), Wollongong, Australia.
- Williams, G. and Clarke, D. (1997): *Mathematical task complexity and task selection*. Paper presented at the Mathematical Association of Victoria 34th Annual Conference - 'Mathematics: Imagine the Possibilities', Clayton, Victoria, Australia.

Integrating Source Code Plagiarism into a Virtual Learning Environment: Benefits for Students and Staff

Thanh Tri Le Nguyen¹ Angela Carbone² Judy Sheard¹ Margot Schuhmacher¹

¹Faculty of Information Technology, Monash University
PO Box 197, Caulfield East, Victoria 3145
tri.lenguyen@monash.edu
judy.sheard@monash.edu
margot.schuhmacher@monash.edu

²Office Pro Vice-Chancellor (Learning and Teaching), Monash University
PO Box 197, Caulfield East, Victoria 3145
angela.carbone@monash.edu

Abstract

Source code plagiarism is a growing concern in computing related courses. There are a variety of tools to help academics detect suspicious similarity in computer programs. These are purpose-built and necessarily different from the more widely used text-matching tools for plagiarism detection in essays. However, not only is the adoption of these code plagiarism detection tools very modest, the lack of integration of these tools into learning environments means that they are, if used, just intended to identify offending students, rather than as an educational tool to raise their awareness of this sensitive problem. This paper describes the development of a plugin to integrate the two well-known code plagiarism detectors, JPlag and MOSS, into an open source virtual learning environment, Moodle, to address the needs of academics teaching computer programming at an Australian University. A study was carried out to evaluate the benefits offered by such integration for academics and students.

Keywords: source code matching tools, academic integrity, plagiarism.

1 Introduction

Ongoing reports of serious plagiarism incidents, as well as a relaxed attitude amongst students towards this offence, have motivated a lot of research to combat this problem from both education-prevention and detection-punishment perspectives (Sheard and Dick, 2011, Dick et al., 2008). A lot of tools have been developed to assist academics with educating their students about plagiarism and detecting this offence should it occur. A very popular example of such a tool is the widely used text matching tool Turnitin, which is not solely intended as a detection tool for instructors, but is also a feedback tool to educate students on academic integrity, with features such as allowing draft submissions and providing students with

access to a similarity report.

Plagiarism involving text (as found, for example, in essays) is the most widely reported form of plagiarism. Plagiarism of computer source code, although less widely reported, has higher incident rates (Sheard et al., 2002, Wagner, 2000). This can be explained by competition amongst students, pressure to create error-free programs, an abundance of readily accessible solutions and a tradition of reusing past assignments (Roberts, 2002). Furthermore, source code is much more constrained than natural languages, which leads to a high degree of similarity between programming assignments as opposed to essays. With the help of modern programming environments, it is remarkably easy to refactor source code by just a few mouse clicks and make it look very different in appearance. This constitutes a substantial challenge for teachers to detect plagiarised code since they typically do not have enough resources to manually check for similarities, unless some students commit the same distinctive mistakes or implement similar unusual approaches.

Some plagiarism detectors intended specifically for source code such as MOSS (Aiken, 1995) and JPlag (Prechelt et al., 2002) can help a great deal in detecting suspiciously similar code, since their algorithms are specifically designed to overcome common disguising techniques. However, these tools are much less known and adopted than their essay plagiarism detection counterparts (Lancaster and Culwin, 2010). As standalone tools, the source code plagiarism detection tools are neither convenient to use, nor able to provide feedback to students, as can be done with an educational tool like Turnitin.

This paper reports on a project to bring code plagiarism detectors closer to their target users with a plugin integrating the two popular plagiarism detection services, JPlag and MOSS, into Moodle, a widely used virtual learning environment. The aim of the project was to promote the adoption of source code plagiarism detectors, not only as detection tools, but also as educational tools, in order to raise students' awareness of academic integrity. This would be achieved by providing students with feedback on the similarity of their work via a report. The pre-evaluation phase of the project investigated needs and issues faced by academics in their current practices. After integrating the two selected

detectors, JPlag and MOSS, into Moodle in a way that addressed academics' needs, the post-evaluation phase investigated the academics' and students' perceptions of the usability, benefits and impacts of the integration via the similarity report generated on a real-life assignment.

The next section of the paper gives some background, on currently available source code plagiarism detectors coupled with their adoption and impact. The following section presents the research method leading to the integration of the plagiarism detectors into Moodle and the investigation of the benefits of this integration. The fourth section presents the result of the research, and the final section concludes the findings of this study.

2 Background

2.1 Definition of source code plagiarism

Plagiarism is one of the most common forms of academic offence, yet the concept of plagiarism is arguably obscure and confused. One of the most cited definitions of plagiarism is from Britannica Encyclopaedia, which regards plagiarism as *"the act of taking the writings of another person and passing them off as one's own"* (Britannica, 2012). When applied to computer source code, an early paper defined plagiarism as *"a program which has been produced from another program with a number of routine transformations"* (Parker and Hamblen, 1989). However, these definitions are not particularly clear and there seems to be no consensus amongst computing academics on one single universal definition which can apply unambiguously to every case (Cosma and Joy, 2008). The reason for this divergence is due to the nature of programming languages which have less freedom and flexibility than natural languages, the ease of code modification, and the encouragement for code reuse. These characteristics make it easy to disguise plagiarised code, or to unintentionally commit the offence, not to mention the fact that offenders can rely on the ambiguity of the rules to claim innocence (Wagner, 2000).

Some research has tried to determine the boundary between the accepted practice and plagiarism in computer programs through surveys of academics and students. However, opinions recorded are diverse, especially in subtle cases where the student's contribution is substantial (Cosma and Joy, 2008). This variation can be explained by the differences in teaching and assessment objectives. Therefore, judgement about plagiarism cannot be made independently of context (Dick et al., 2008).

2.2 Source code plagiarism detectors

While essay plagiarism detection tools search for consecutive words to identify copying, source code plagiarism detectors have to take into account the modifications that students can make to disguise their code. Faidhi and Robinson (1987) characterised six levels of modification from simple to complicated. From the original program (level 0), plagiarists could change comments and indentations (level 1), identifier names (level 2), declaration of constants, variables and procedures (level 3), program modules (level 4), program statements (level 5) and finally logic expressions (level 6). With the help of modern programming

environments, modifications at level 3 and below only require very primitive knowledge of a programming language, while it is controversial that level 6 could even be considered as plagiarism (Cosma and Joy, 2008).

Different kinds of source code plagiarism detectors are able to find copied code at different levels, depending on the type of algorithms they use. There are basically three categories of detection tools in this regard: software-metric based, token-based and semantic-based. Software-metric based detectors, the most primitive ones, use metrics in software engineering such as program size, complexity, as well as the number of keywords, operators and operands to compare program code (Parker and Hamblen, 1989). The token-based method generally finds common consecutive tokens between two programs after eliminating possible variations such as identifier names (Prechelt et al., 2002). Semantic-based detectors, the most sophisticated ones, construct parse trees or program graphs for each piece of code before matching them together (Liu et al., 2006). While the last category of detectors may be able to detect plagiarised code at the highest level, they do not scale well for large sets of programs, and no publicly available tools in this category were found. For student assignments, token-based tools, which encompass most of the popular detectors, are considered to be the most appropriate for educational use.

2.3 The adoption and effects of plagiarism detection tools

Reports suggest that the use of source code plagiarism detectors is quite limited compared to the scale of the problem and the adoption of these tools is much less than their natural language counterparts (e.g. Turnitin). In fact, only one quarter of the institutions in UK reported having adopted code plagiarism tools (Lancaster and Culwin, 2010). As a result of this lack of use, when students were suddenly checked for plagiarism in their programming assignments, the offences were found to be unexpectedly high (Wagner, 2000, Daly and Horgan, 2005, Bowyer and Hall, 2001).

Whilst some papers have reported studies of the effects of using a text matching tool on students' text-based plagiarism practices (Biggam and McCann, 2010, Rolfe, 2011), not as much research has reported on the same issue with source code plagiarism. Bowyer and Hall (2001) observed that the incident of plagiarism dropped considerably after a period of time using MOSS, but later discovered that plagiarism was shifted to another source undetectable by software such as acquiring a program from students from a previous running of the course or even hiring outsiders. The problem was only detected when two students in the same class acquired the same program from one source, which the authors called the *"ghost author phenomenon"*. Similar incidents were also reported at RMIT University where JPlag is used officially for every programming assignment (Zobel, 2004, D'Souza et al., 2007).

Besides detection, a suggested educational benefit of the detectors is the raising of students' awareness of the offence by providing them with feedback on the similarity of their code to others. A search in the literature found no works that have been done to evaluate the

educational effects of this kind of feedback from source code matching tools. However, similar research on Turnitin by Biggam and McCann (2010) found that this feedback could facilitate a smoother transition between school and university for first year students by helping them to improve their referencing skills and awareness of academic integrity.

3 Research approach

The widespread problem of plagiarism in programming assignments, as opposed to the low adoption of source-code plagiarism detectors, has motivated this research to seek a better understanding of the benefits and limitations of these tools and promote their adoption. The project proceeded in three phases:

- Phase 1: Investigation of the current plagiarism detection and handling practices of academics and their satisfaction with these practices.
- Phase 2: Development of a plugin to integrate the two well-known third party plagiarism detection tools MOSS and JPlag into Moodle.
- Phase 3: Evaluation of the benefits of the plugin in making MOSS and JPlag code plagiarism scanning services an inherent part of the assessment process within Moodle. This will be conducted with a real assignment and from the perspectives of both academics and students.

The remainder of this section describes each phase, explaining the purpose of the phase, a justification of the approach, the method involved and its potential limitations.

3.1 Phase 1– Investigating the current practice

In order to integrate the detectors in a manner that meets academics' needs, it was important to obtain insights into the current practices of plagiarism detection and the difficulties encountered. The aims of this phase were twofold:

- *Gain understanding of the current practices in identifying and dealing with source-code plagiarism:* this was the main focus of this phase since it helped determine potential enhancements of the integration of the detectors into the virtual learning environment the academics used.
- *Determine the level of satisfaction with the current practices and the difficulties encountered:* this would highlight areas of improvement to academics' assessment practices that the tools may offer.

Given the exploratory nature of this phase, a semi-structured interview approach was adopted. The interview method was considered over the other qualitative methods, such as surveys and focus groups, due to its flexibility in exploring the academics' views. While a survey could reach a larger group of participants in a limited time, it does not offer the opportunity to delve into the ideas raised by academics, which were of particular interest to the research. Moreover, interviews make it easier for respondents to provide their opinions and personal experiences, which were actively sought in this phase.

The study was conducted with Monash University academics, including both lecturers and tutors, who were involved in teaching programming units at any level. Twenty-two academics were approached for interview. This was done by sending an email invitation to lecturers and tutors of programming units. Every academic who agreed to take part in the research was interviewed.

A limitation of interviewing only Monash academics is the lack of representativeness for the whole IT academic community in general. Since academics from the same university often adhere to the same standards and processes, they might be likely to share similar views and practices. This potentially results in the findings of the research and the implementation of the plugin having less applicability to other institutions.

3.2 Phase 2 – Developing a Moodle plugin

Two popular source code plagiarism detectors, MOSS (Aiken, 1995) and JPlag (Prechelt et al., 2002), were identified as having good detection performance and good reputations in the academic world. Taking into account the needs and difficulties raised in the previous phase, the aim of this phase was to develop a Moodle 2 plugin for the integration of these two detectors with the following characteristics:

- *Seamless assignment creation and submission:* the use of these plagiarism scanning services should be effortless and transparent to the users, without imposing any additional constraints to the normal submission process.
- *Publishing a limited version of the similarity report to students:* seeing plagiarism detectors also as an educational tool and a means of feedback to students, the plugin would allow lecturers to publish a limited version of the similarity report on the students' code, whilst ensuring confidentiality by masking identities.

3.3 Phase 3 – Evaluating the plugin

The aim of Phase 3 was to evaluate the plugin based on academics' and students' opinions of the similarity report generated from a live assignment. It was considered that using a live assignment rather than artificial data would make it easier to elicit feedback from lecturers and, in particular, students, since a live assignment is more natural and provides a specific context. Academics and students would be given the report to view in order to provide feedback in an interview. The following were the issues that the interview focused on:

- The academics' comments on the usability of the similarity report and any improvements it offers to their assessment process.
- The students' opinions about the value of the feedback on their work given by the generated report and its impact on their ethical conduct.

A medium size programming assignment was deemed appropriate for trialling the tool. The first assignment of a Java unit¹ was selected because of the suitability of its

¹ A unit of study at Monash is equivalent to a subject in other university

schedule to the research timeframe. This unit is a foundation programming unit for post-graduate students. More than 20 students in the unit were introduced to the research project in a lecture and invited to submit their assignment work to a Moodle sandbox. The sandbox was separated from the University virtual learning environment so that the confidentiality of students' work could be ensured. As plagiarism is a sensitive topic for some students, at the time of participation, students were assured that their marks would not change and the generated report would not be shown to their lecturer and tutor. The incentive for students to participate in the research was that they would have the opportunity to see the report and give comments about it.

Academic participants in this phase were recruited from the participants in the first phase. In the first interview, academics were asked whether they were willing to evaluate a similarity report in the latter phase of the research. Everyone who gave consent was recontacted in this phase.

Interviews were conducted individually and consisted of open-ended questions. Participants were free to express their opinions and relate their experiences. Interesting issues raised by participants were explored more deeply. All interviews with academics were recorded and transcribed. However, only hand written notes were taken during the student interviews to remove any possible discomfort with the recording.

4 Results

Following the research approach described above, this section details the findings collected through the interviews in phase 1 and phase 3. Recurrent themes were extracted from the transcript and classified into emergent categories.

4.1 Phase 1- Investigating the current practices

Six academics agreed to participate in this research. It was found that the academics participating in this study followed very similar practices of dealing with plagiarism, although their satisfaction towards the effectiveness of these practices varied.

4.1.1 Academics' practice of dealing with plagiarism

The study found that most academics deal with plagiarism more on the prevention side than the detection side. Most participants at the time of interviews did not have a formal process to detect plagiarism, while they mentioned many methods on the prevention and education side.

4.1.1.1 Plagiarism prevention

Prevention strategies were raised the most during the interviews. The following are the approaches mentioned by academics.

Assignment design

Most academics interviewed affirmed that they spent considerable effort to make sure that their assignments are not easy to find on the Internet, thus making them more difficult for the students to plagiarise:

"I'm happy that my assignment is not adapted from somewhere else where [students] can get the code" ... "My responsibility as a lecturer is to create an assignment that is difficult to plagiarise. I can't always blame the students."

In regard to the design of assignments, the academics felt it is preferable that an assignment is not too small and has a variety of possible approaches and designs, so that students who work independently would produce significantly different code. This would also reduce the temptation to plagiarise and make it easy to recognise striking similarities later:

"In my assignment, most of the stuff is very unique and most of the time students come up with a unique solution"

Exam weighting

Another method to reduce the desire to plagiarise is to give the exam a much higher weight than the assignments, since it is very difficult to cheat during an exam. The intention is to encourage students to work on the assignment to acquire the skills needed to succeed in the exam. This type of mark distribution is used in introductory programming units, where assignments are quite simple and students' code is likely to be similar to each other:

"It's hard to [pass] because the exam has the majority of the mark. If they do cheat in the assignment, [...] they are not able to pass anyway"

Avoiding assignment reuse

Most participants emphasised the need to avoid reusing the assignment over many semesters:

"If I use the same assignment over and over, then I am inviting people to do plagiarism because I know that some of the students know people who did the subject last year or the year before. I always make changes to the assignments every year, so they can't copy and paste from last year's students without understanding".

Nevertheless, the extent of variation differed between academics. One participant claimed that he changes his assignments completely each semester, whereas another reported that sometimes he reused old assignments. The majority stated that they make some variations but the main ideas and concepts remain unchanged. Depending on the extent of variation, this strategy may only prevent blind copying. Since the assessed concepts were the same, students could adapt a past year's program with much less effort than doing it from scratch:

"If they have access to [last year's] student work, it would be helpful to them because the concept is the same. But they cannot copy and paste"

Considering the effectiveness of these efforts, academics also maintained that what they do would just reduce the problem of students copying and pasting from each other. However, there are always other sources for students to plagiarise from:

"Students can post questions on forums and, sometimes, get the whole program from others"... "Using the Internet, students can hire others to do an assignment for them... Sometimes, they can outsource it to India..."

Tutorial assistance

A couple of participants mentioned following up with students during tutorials in order to provide adequate feedback. As one explained:

“In every tutorial, the students must complete their work and submit a lab sheet each week. Every week, I check the lab sheet and give them feedback. So I get a fairly good idea of what each individual is capable [of]”

Besides identifying students who need help and assisting them early so that they can do the assignment by themselves, knowing the students’ levels of performance also helps academics know who to pay attention to later when investigating suspicious plagiarism cases.

4.1.1.2 Plagiarism education

Raising students’ awareness of plagiarism is important, especially in the area of source code where the boundary between plagiarism and proper practice is vague. Every academic in the study claimed that they often remind their students about plagiarism. Some stated that they explained their expectations before an assignment:

“I made it quite clear to all the classes that we are happy for them to discuss the assignments, but the implementation – that has to be their own. Be prepared to justify your design in the interview... I also posted something more on the discussion board.”

Others, however, were less pro-active and just mentioned it as a standard procedure in a very general manner:

“I don’t know that I made it explicit. I always say to them that the work that you submit must be your own work, not someone else’s. They might find it’s silly since I am saying the obvious”

4.1.1.3 Plagiarism detection

When it comes to plagiarism detection in programming assignments, the methods that every participant used were either program demonstration in the lab and/or in-depth interview with the students:

“I detect plagiarism through a demonstration. Whatever they submitted to Moodle, I ask them to download and run it. Then I go to a particular class, and point to a complicated piece of code and ask them to explain it. I also ask more conceptual question, such as how did you do it or why do you use this approach.”

However, interviewing is not possible in every situation. One academic said that he could not rely on interviewing since many of his students were not able to meet face to face for interviews:

“Many students of mine are distance learning. So many times I don’t interview.”

Other than interviewing or lab demonstration, most participants stated that they also read the code for marking and giving feedback, and during this process plagiarism cases are sometimes uncovered when students do weirdly similar things or commit the same mistakes:

“I also read the code of the students fairly thoroughly because I have to give feedback. When I

read the code, sometimes I pick up something very unusual and another student did exactly the same thing or made the same error. Then I compare the code to see if it has too much similarity.”

If the academic monitors students closely in tutorials, they can target some students who they observe working together in groups:

“Because we know the students quite well, we would tend to know who are working together and we could start to identify this during the process of marking. We do a little bit of cross checking to see how they are similar.”

or, they can target students whose performance in the assignment differs too much from what they have demonstrated in labs:

“If they produced exceptionally good work and I know that this student had trouble with programming in the tutorials, then I know usually something is going on”.

With regard to the use of technology, it was found that only one participant had used a technology to detect plagiarism:

“I sometimes use the diff command in Linux to compare two codes where I found some similarities.”

Another participant had previously used a tool to detect similarities in Java code, but considered its overhead not to be worth the value that it brought:

“From my previous experience, I am not very convinced that software can do a good job. Software can just pick up blatant similarity. Then, if it is so obvious that software can pick up and I can pick it up myself, then going through the hassle of setting the software, and feeding all the assignments to it, is not worth the trouble.”

The remaining participants explained that they did not used tools because they were not aware of any that were suitable:

“I didn’t use any formal plagiarism detection tools... I haven’t found out about them yet.”

or because they thought these tools were unnecessary because interviewing alone could ensure that the students actually understand what they submit:

“I can be pretty sure that students don’t get the mark for what they don’t learn... It’s probably [too] much trouble to examine what comes back from a plagiarism detector.”

Another reason was that the purpose of plagiarism detectors seems to be orthogonal to the purpose of teaching and learning:

“Using plagiarism detection is something that is essentially a punishment. That is probably not a core point I think. It’s far more important to use that opportunity to learn these things and improve themselves.”

In brief, when it comes to plagiarism detection, academics just follow the normal process of assessment: using demonstration and interviews to check the students’ understanding and reading code to give feedback or a mark. There are no separate stages that are devoted solely to detecting plagiarism. Plagiarism was discovered in an

ad-hoc manner when some unusual similarities or striking progress captured the marker's attention. At the time of the interviews, none of the participants were using code plagiarism detectors, although one participant had tried one in the past but found that it posed too much overhead on the assessment process.

4.1.2 Difficulties and satisfaction with the current methods of detecting plagiarism

By understanding academics' satisfaction with current plagiarism detection practices and difficulties they encounter, possible improvements could be identified. It was observed that their level of satisfaction and the difficulties reported depended largely on their degree of tolerance towards plagiarism, despite their similar approaches to plagiarism detection.

4.1.2.1 Level of satisfaction with the current plagiarism detection method

Academics who believed that assessing the students' learning and understanding is sufficient claimed that their current approach is satisfactory:

"If the students know the stuff as much as I know, there is little point in searching for plagiarism... Students in the interview, they can't get away with it because they don't know it... [With what I currently do], I can be pretty sure that the students don't get the mark for what they haven't learned."

On the other hand, academics who took copying seriously expressed their discontent with their current practice, since they felt certain that some plagiarism cases escaped detection:

"It's possible that the students can get somebody else to do their work and really study well and understand it so that during the interview they can explain it... I'm not satisfied in the sense that I'm sure there are some cases that go unnoticed."

However, these academics also affirm that what they have done is adequate within the limited time they have

"But given all the constraints on time, I don't feel that I should put a whole lot more effort to detect plagiarism. Overall, I think it's reasonable"

4.1.2.2 Difficulties encountered

Class size

Large class sizes were one of the major difficulties raised by academics. With a class of hundreds of students, it is merely by chance that cases of plagiarism are detected:

"It depends on how many students I have. A few years ago, I had hundreds of students. It was lucky to find some plagiarism cases since there were so many students. But if there are only a few, it is very easy"

and also, there are many markers who grade the assignments independently, which further reduces the chance of detection:

"I also read the assignments and sometimes pick up some similarity. But it doesn't ensure anything because we have many classes and markers"

Time constraints

Another difficulty encountered by many academics was the limited time they could spend on assessing each student, for both the interview and the marking:

"We have a two hour lab. I don't want to spend more than 15 minutes per student just trying to detect plagiarism"

Therefore, some academics admitted that they sometimes did not want to spend more time finding similarities, even when in doubt:

"Even if I remember all this, for me to go back to find it takes too much of my time. If I know who their friends are, I just look through those ones, but if I couldn't find [anything], I probably wouldn't go through too many assignments, just to find something I thought I remember that is similar."

Low variation between assignments

There are some kinds of assignments for which students' work is inevitably similar. For example, when students were given skeleton code or only required to adapt code given in their lab. In these cases, it is not possible to tell if the students plagiarised, as one academic pointed out:

"The point is that the size of the program is often not big enough to be very distinctly different, and in more advanced units, we actually give them code so they will be very much the same."

However, all other participants claimed that the way of implementation should be different even when the same approach was taken:

"In my assignment, there are a few ways that are quite clear that they are the best ways to approach it... I still expect some deviation in the way classes are implemented or the logic on how it works"

Overall, the academics in the study thought that their approaches were effective in reducing plagiarism, although they admitted that some cases of plagiarism can slip through the process undetected. For academics that were happy with assessing students based only on the students' understanding, this process was satisfactory. However, those who wanted to make sure that the students actually did the work themselves thought that what they currently did was not enough due to time limitations for marking and interviewing, as well as the difficulty of manual checking in large classes where there are many markers.

4.2 Phase 2 – Building a Moodle plugin

4.2.1 The need for integrating the detection tools into Moodle

It was found in the previous phase that academics were concerned about plagiarism and paid attention to minimising the motive and temptation to plagiarise amongst their students. Furthermore, they tried to detect plagiarism in their marking process. However, due to the constraints of time and resources, most academics acknowledged that discovering plagiarism was quite ad-hoc and accidental, relying on unusual similarities, which captured their attention. The adoption of a good plagiarism detector could make this process less dependent on chance.

The two major difficulties raised by academics in detecting plagiarism, large class sizes and time limitations, could be mitigated by an automated process using the tools. Moreover, the detectors could make cross checking between markers much easier by generating a single report across the whole class.

However, as one participant remarked, code similarity detectors, when used as stand-alone tools, impose considerable overhead. The repetitive tasks of downloading and extracting all students' submissions and then organising them into the required directory structure are time consuming and error-prone. This is compounded by the non-intuitive interface making the adoption curve steeper. Moreover, these plagiarism detectors, when used as stand-alone, are "*essentially a punishment*", as one academic thought. It cannot serve as a means of providing feedback for students and raising their awareness of academic integrity.

Considering all the reasons above, it seems reasonable to propose that integrating the detectors into a virtual learning environment like Moodle could significantly promote their adoption. It would enhance the usability of the tools by providing an intuitive interface in the learning environment familiar to academics. In addition, it would make the similarity scanning transparent and effortless, at the same time allowing academics to provide students with restricted access to the similarity report.

4.2.2 Features of the plugin

Considering the needs and difficulties mentioned above, the plugin was developed with features that make the detection tools fit seamlessly into the assessment process on Moodle, namely:

- *Automatic filtering of code files*: students can submit an archive file containing a bunch of project files and documentation apart from code. The plugin will extract only code files to send them to the scanning service. Therefore, no additional constraints are imposed on the system other than the normal submission and marking process.
- *Automatic scanning and reporting*: the assignments are extracted and submitted automatically to the scanning services at the date specified by the lecturer when configuring the assignment.
- *Publishing similarity report to students*: the plugin enables teachers to allow students to view the report on the similarity of their code with others'. If permitted, students could only see a restricted version of the report, with names masked.

Currently, our system keeps the native reporting interfaces of MOSS and JPlag. The MOSS interface presents a simple list of pairs in decreasing similarity order, whereas JPlag groups all the students having high similarity with one student in a row. In addition, JPlag also computes the similarity distribution of all the pairs.

4.3 Phase 3 – Evaluation of the plugin

The live assignment selected to evaluate the plugin involved the development of a small "number guessing game" in which the user and the computer took turns to guess a random number within a specified range. Fifteen

students agreed to take part in the study and submitted their code to our sandbox. Most of them were international students, and many were in their first semester of study.

4.3.1 Overview of the generated report

All of the 15 submissions were of similar structures. The students followed the design they had been recommended in their lab, with a Player and a Game class, which were used for storing the information of the player and controlling the game logic respectively. Depending on how carefully the students handled input errors and how much boilerplate code (e.g. set and get methods) they wrote, the length of their submissions varied. Excluding comments and blank lines, the average number of lines of code of all submissions was 193. The generated similarity report showed the pairwise similarity rate among 15 submissions ranged from 0% to 35%. Overall, MOSS gave considerably lower percentages than JPlag. The similarity distribution given by JPlag was symmetrical with half of all the pairs having similarity rates in the middle range (10%-20%), a quarter of the pairs in the lower range (0-10%) and the other quarter in the higher range (20-30%), plus three "*exceptional*" pairs having similarities distinctly higher than the others. MOSS produced a very different distribution with most of the pairs (70%) having similarity rates below 10%, the other 30% of the pairs having rates ranging from 10%-20% and one "outlier" pair had an outstanding rate of 20.5%. For a small and strictly specified assignment such as this, the similarity rates were considered quite low.

4.3.2 Evaluation: academic perspective

Four academics who participated in the first phase were reinterviewed in this phase. In addition, we extended our invitation to the Moodle administrator of the faculty, who is also experienced in teaching programming, to get a more varied perspective on the plugin. In the Phase 1 interview, academics expressed different opinions with regard to using detection tools, from a high level of interest to doubts about their benefits. This phase reinvestigated their views after introducing them to the plugin and presenting them with the similarity reports. Overall, academics expressed favourable opinions on the plugin and their interest in using it in a further trial.

Improvement of the plugin to the assessment process

Academics all had a positive first impression of the plugin. A variety of enhancements to the marking process mentioned by academics included time efficiency, reduced effort, more stable detection and better student awareness of academic integrity.

Although every academic agreed that the plugin cannot automate the detection process entirely, most academics affirmed that the readily available detectors could save them a lot of time and effort, not to mention better detection effectiveness:

"What I used to do when I mark... if I think it's plagiarism, I put a mark at the top [...]. At the end, I tried to figure out which one is similar to which one. It's very time consuming, especially when I have 50-60 [students]. I have so many things to do. Most of

the time, I just try to go to as much as possible [until] I give up. A tool like this makes life much easier.”

However, a couple of other academics also mentioned the extra time they would need to spend examining the reports:

“It actually takes more time to examine the report. But it helps better ensure academics integrity. And while checking the report, I also read the students’ code. So, it takes less time to review the code”

Some academics emphasised the usefulness of the plugin for large classes with many tutors who mark separate groups of students:

“When I have one class with a hundred students, there are often four to five tutors. It’s not easy for the tutors to sit together to do the [cross checking]. A tool like this one would be handy”

A feature of the plugin that attracted opposite opinions is publishing the report to students. Some academics commented that it could serve as an effective plagiarism deterrent:

“It’s OK for the system to say we ran the scanning through and we don’t find any similarity. It may alert them and stop them from plagiarising.”

Nevertheless, many concerns were raised. A major issue is the potential anxiety of students when their code contains a high degree of coincidental similarity:

“If I tell the students who may not plagiarise that there is 30% similarity between me and you, it may make the students anxious.”

Moreover, this may result in a negative impression among the students, making them think their lecturers are primarily interested in catching them:

“We really don’t want students to think that we’re doing a switch hand, that we are having the system to drag them into trouble”

These concerns were further investigated with students’ perspectives on the plugin. Some academics preferred to have a fine-grained control over the publication of the scanning results to students. For example, they wanted to filter what reports were sent out to students, particularly those that were above a plagiarism threshold.

“It’s better if the system lets the lecturers to decide to post to the student one by one”

Usability of the similarity report

All five academics agreed that the report was very easy to read. They commented that the interface and design of the report helped them to quickly identify suspicious cases:

“I am really happy with the interface. I could navigate between the similarities very quickly”

There were somewhat different comments on the reporting interface of MOSS and JPlag. More academics preferred the tabular style of JPlag listing all students similar to each student on a row than a simple list of pairs in MOSS:

“[The interface is] fine with MOSS. But JPlag is better. With this style of presentation, I can see all the students who have a high similarity with one student... It makes the examination easier”

Nevertheless, one academic expressed a different opinion:

“JPlag is a bit more confusing... Not all the students having a high similarity rate appear on the right hand side. It’s more confusing to follow than the simple view of MOSS”

A major benefit of JPlag over MOSS expressed by the academics was the statistics of similarity distribution between every pair:

“The statistics help me to see immediately a few suspicious cases, those who have much higher similarity than the average”

Generally, the additional features provided by JPlag over MOSS were appreciated by all the academics.

Satisfaction with the tool

Overall, participants were positive about the plugin and expressed interest in trying it in their units, though each to a different extent. Some participants expressed their satisfaction with the report:

“I am impressed by this. The tools clearly told me what the similarities are... all the things according to the highest to lowest”

Others expressed a more reserved opinion on the tool’s usefulness, and expressed their desire to experiment with the plugin in their units:

“The answer now is yes. I think it is useful, but until I try it I can tell how useful it is... Yes, definitely. I am curious about it and want to try”

Overall, academics expressed satisfaction with the plugin and the detectors after experimenting with its features and examining the report. They felt the plugin offers many benefits, including reducing effort in plagiarism detection, more effective use of time, and improved detection effectiveness especially in large classes. Publishing the report to students was generally supported, although there were some concerns about students’ anxiety and confidentiality.

4.3.3 Evaluation: student perspective

This section presents the opinions of students on the report and its influence on their ethical behaviour. Among the 15 students who had initially submitted their assignments for the research, three were interviewed in this study. One of them had a little programming background before joining the unit, and the other two were learning programming for the very first time. These students all had the detected similarity rate a bit higher than the average (ranging from 25% to 31%). However, a look through the similarity report showed that they were unlikely to have copied code from each other.

Students’ opinion on the similarity report

All the students stated that the similarity rate of their work was higher than they expect, although they agreed that the tools were quite accurate in highlighting the similar portions in their code. For example, one student expressed that he did not expect such a high similarity since he did the assignment alone. However, he thought that the tools picked up the interesting similarities.

Another student stated that they should be assessed based on their understanding of what had been done, not

based on the similarity rate with others, since it is very easy for assignments to be similar.

A common concern raised was that students thought that coincidental similarity is inevitable and that such a report on their work may be misleading.

It was observed that a higher than expected similarity caused certain anxiety amongst participants. After the report was released, one student sent an email justifying his incorrect similarity rate (which was just 24.3% and lower than the interviewed students), by pointing out the false positives in the report. This student did not take part in the interview.

"I [have] just seen the result of your system. My code was [reported] as having a 24.3% similarity with another student and I think it's not right... For example, the group of println statements in lines 41-48 is very different, but marked as similar... The similarity score is higher than reality... I worked independently in this assignment"

As a contrast, another participant stated that he thought his similarity rate was normal, though he had the highest similarity rate amongst the three students interviewed (30.9%). He was confident that the examiner would see that there is no sign of plagiarism in his assignment. This student said he had had experience with an essay plagiarism detector in his undergraduate study.

Although participants were told before joining the research that the report was confidential and would not affect their result, and that there were many reasons for code to be similar, anxiety still occurred. It seems that anxiety does not only depend on the similarity rates found but also on the students' prior experiences with such a tool. If students have been exposed to a similar tool before, they are likely to feel more comfortable in how the results would be interpreted, even if the similarity is quite high.

Impact of using the plugin on students' behaviours

The participants indicated that the strongest impact of using the plugin would be in discouraging them from sharing code with classmates. All of the students said that they *"will not dare to share code with anyone"* since there is little chance to escape the detectors.

As for sharing on a conceptual level, opinions of students diverged. Some students said that they would be even afraid of sharing ideas and approaches since the chance of coming up with similar code would be much higher. However, the student who had had some programming knowledge prior to joining the unit, said that he would not feel anxious about sharing his algorithm or method in a general way, since it is not likely that there would be a similar implementation based on just the ideas.

All students interviewed said that they would be afraid of copying code from another student since there is a good chance for them to be detected. One student stated that if he copied the code from someone else, he would have to modify the code substantially to avoid being detected. This job requires time and understanding of someone else's code; therefore, he concluded that it is better and safer to write his own program.

4.3.4 Overall summary

The evaluation of teachers' and students' perspectives on the Moodle plugin showed that it offers appropriate features to assist academics in detecting plagiarism. The plagiarism detectors are quite efficient and potentially effective in discouraging students from plagiarising. The academics stated that the plugin was easy to use and the report interface allows for faster plagiarism detection than manual scanning. Although, more time must be spent in examining the report, this time could be offset by less code reading time and better detection. The plugin is especially useful for a large class with many markers, where manual cross checking is often not possible. The students stated that plagiarism detectors would deter them from copying and discourage them from collaborating. However, using the tool also produced some undesirable effects. When students know that such tool is being used, it may also cause anxiety amongst students and hinder some forms of exchanging of ideas that should be encouraged. The extent of anxiety would depend on their exposure to other plagiarism detectors and their programming experience.

Publishing the report to the students was generally supported by the academics, provided that confidentiality is maintained and the results are explained to the students. However, there were different opinions from the students. One student considered that the report is part of a student's result and they should be allowed to see it, whilst another student did not want their code to be revealed to others.

5 Conclusion

This paper described a study to investigate the benefits of source code plagiarism detectors for current assessment practices and to promote their adoption by integrating two well-known detectors into Moodle via a plugin. The plugin was built to respond to the difficulties encountered by academics in their current practices and its usefulness and impact were evaluated from both lecturers' and students' perspectives.

Whilst academics devote considerable attention to the issue of plagiarism, with a range of measures for prevention and detection, many of them were unsatisfied with their current practice since they were quite certain that some cases of plagiarism went unnoticed when their class size was large and their time to spend on marking was limited. All participating academics did not currently use any plagiarism detection tools for code, as they had not investigated any tools or they found that tools posed considerable overhead.

Taking into consideration all of the difficulties expressed by the academics in this study, a plugin to integrate JPlag and MOSS into Moodle was implemented. The plugin makes these detectors effortless to use and enables the scanning results to be made available to every marker. In addition, it offers the possibility of giving feedback to students and therefore has value as an educational tool.

Evaluation from the lecturers' and students' perspectives showed that the tools could assist academics effectively in detecting plagiarism, and deter the students from sharing code together. Making the similarity report

public to the students was considered a good idea by most of the academics as they saw it raising students' awareness of plagiarism although there were some concerns about confidentiality and the students' levels of anxiety. In fact, interviews with students showed that their first exposure to a plagiarism detector might arouse significant anxiety, since non-plagiarising students expected a negligible similarity, which often was not the case. This anxiety may also vary according to the students' programming experience and their understanding of the process.

The evaluation of our current plugin has also revealed a lot of room for improvement. Our current system uses the native interfaces of JPlag and MOSS, with just a few modifications to hide identities in the student view of the report. The feedback from this study indicates that it is better to incorporate the result of different engines into a single interface which combines the advantages of each, helping academics browse the results and filter suspicious cases faster. The draft submission is also a very interesting feature that is available in some text matching systems. Our current plugin permits students to see the final report, but does not offer the option to resubmit their work if they find their similarity rate is high.

A major limitation in the research method is that academic participants are just given a demonstration of the plugin before evaluating it instead of actually experimenting with it themselves. This, perhaps, limits the richness of the information they could give.

With these identified shortcomings in the research method and the plugin, future directions of our research are to improve the plugin and to re-evaluate it by giving participants hand-on experience of configuring and using the tool.

- Aiken, A. 1995. *MOSS - A System for Detecting Software Plagiarism* [Online]. Available: <http://theory.stanford.edu/~aiken/moss/> [Accessed].
- Biggam, J. and Mccann, M. (2010): A study of Turnitin as an educational tool in student dissertations. *Interactive Technology and Smart Education*, **7**(1): 44-54.
- Bowyer, K. W. and Hall, L. O. (2001): Reducing Effects of Plagiarism in Programming Classes. *Journal of Information System Education*, **12**(3): 141-148.
- Britannica 2012. plagiarism. *Encyclopædia Britannica Online*. Retrieved 01 November, 2012, from <http://www.britannica.com/EBchecked/topic/462640/plagiarism>.
- Cosma, G. and Joy, M. (2008): Towards a Definition of Source-Code Plagiarism. *IEEE Transactions on Education*, **51**(2): 195-200.
- D'souza, D., Hamilton, M. and Harris, M. C. (2007): Software development marketplaces: implications for plagiarism. *Proc of the ninth Australasian conference on Computing education*, Ballarat, Victoria, 1273676, **66**: 27-33, Australian Computer Society, Inc.
- Daly, C. and Horgan, J. (2005): Patterns of plagiarism. *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, 383-387, ACM.
- Dick, M., Sheard, J. and Hasen, M. (2008): Prevention is Better than Cure: Addressing Cheating and Plagiarism Based on the IT Student Perspective. In: *Student Plagiarism in an Online World: Problems and Solutions*. Roberts, T. S. (ed.). Hershey, PA : Information Science Reference.
- Faidhi, J. and Robinson, S. (1987): An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, **11**(1): 11-19.
- Lancaster, T. and Culwin, F. (2010): A Comparison of Source Code Plagiarism Detection Engines. *Computer Science Education*, **14**(2): 101-112.
- Liu, C., Chen, C., Han, J. and Yu, P. S. (2006): GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. *KDD '06 Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* New York, ACM.
- Parker, A. and Hamblen, J. O. (1989): Computer Algorithms for Plagiarism Detection. *IEEE Transactions on Education*, **32**(2): 94-99.
- Prechelt, L., Malpohl, G. and Philippsen, M. (2002): Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science*, **8**(11): 1016-1038.
- Roberts, E. (2002): Strategies for promoting academic integrity in CS courses. *32nd Annual Frontier in Education*, **2**: F3G-14.
- Rolfe, V. (2011): Can Turnitin be used to provide instant formative feedback? *British Journal of Educational Technology*, **42**(4): 701-710.
- Sheard, J. and Dick, M. (2011): Computing student practices of cheating and plagiarism: a decade of change. *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, Darmstadt, Germany, 1999813, 233-237, ACM.
- Sheard, J., Dick, M., Markham, S., Macdonald, I. and Walsh, M. (2002): Cheating and Plagiarism: Perceptions and Practices of First Year IT Students. *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, New York, ACM.
- Wagner, N. R. 2000. *Plagiarism by Student Programmers* [Online]. Available: <http://www.cs.utsa.edu/~wagner/pubs/plagiarism0.html> [Accessed].
- Zobel, J. (2004): "Uni cheats racket": a case study in plagiarism investigation. *ACE '04 Proceedings of the Sixth Australasian Conference on Computing Education*, Darlinghurst, Australia, **30**, ACM.

Author Index

- Ahadi, Alireza, 87
- Cain, Andrew, 127
- Cameron-Jones, Michael, 31, 51
- Carbone, Angela, iii, 41, 145, 155
- Caspersen, Michael E, 137
- Ceddia, Jason, 41
- Charleston, Michael, 77
- Chinn, Donald, 145
- Chinthammit, Winyu, 31, 51
- Clear, Tony, 145
- Cooper, Graham, 23
- Corney, Malcolm, 87, 145
- Crane field, Stephen, 3
- D'Souza, Daryl, 41, 117, 145
- De Salas, Kristy, 51
- de Salas, Kristy, 31
- Dermoudy, Julian, 31, 51
- Ellis, Leonie, 31, 51
- Falkner, Katrina, 107
- Falkner, Nickolas, 107
- Fenwick, Joel, 145
- Fidge, Colin, 97
- Gasson, Joy, 13
- Gluga, Richard, 77
- Haden, Patricia, 13
- Haig, Thomas, 107
- Hamilton, Margaret, 117
- Harland, James, 117, 145
- Herbert, Nicole, 31, 51
- Hogan, James, 97
- Hu, Minjie, 3
- Kasto, Nadia, 59, 67
- Kay, Judy, 77
- Laakso, Mikko-Jussi, 145
- Lewis, Ian, 31, 51
- Lister, Raymond, 77, 87, 97
- Mason, Raina, 23, 41
- Nguyen, Tri Le, 155
- Nowack, Palle, 137
- Parsons, Dale, 13
- Schumacher, Margot, 155
- Sheard, Judy, 145, 155
- Simon, 41, 77, 145
- Springer, Matthew, 31, 51
- Teague, Donna, 77, 87, 145
- Whalley, Jacqueline, iii, 59, 67
- Winikoff, Michael, 3
- Wood, Krissi, 13
- Woodward, Clinton, 127

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 134 - Data Mining and Analytics 2012**
Edited by Yanchang Zhao, Department of Immigration and Citizenship, Australia, Jiyong Li, University of South Australia, Paul J. Kennedy, University of Technology, Sydney, Australia and Peter Christen, Australian National University, Australia. December 2012. 978-1-921770-14-2. Contains the proceedings of the Tenth Australasian Data Mining Conference (AusDM'12), Sydney, Australia, 5-7 December 2012.